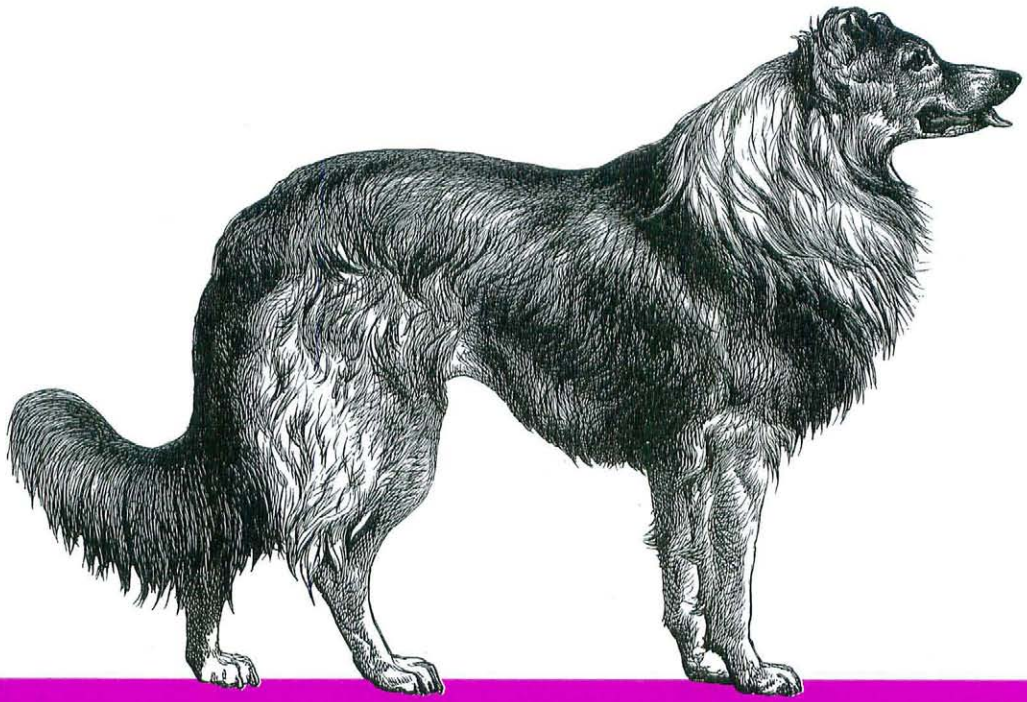
Solutions and Examples for C++Programmers



C++ Cookbook™ 中文版

D. Ryan Stephens, Christopher Diggins, Jonathan Turkanis & Jeff Cogswell 著
金名 周成兴 等译

O'REILLY®
清华大学出版社

C++ Cookbook™

中文版

www.TopSage.com

C++ Cookbook™

中文版

D. Ryan Stepbens, Christopher Diggins,
Jonathan Turkanis & Jeff Cogswell 著
金名 周成兴 等译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc. 授权清华大学出版社出版

清华大学出版社

www.TopSage.com

Copyright ©2006 by O'Reilly Media, Inc.

Authorized Simplified Chinese translation edition, by O'Reilly Media, Inc., is published by Tsinghua University Press, 2007. Authorized translation of the original English edition, 2006 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

本书之英文原版由 O'Reilly Media, Inc. 于 2006 年出版。

本中文简体翻译版由 O'Reilly Media, Inc. 授权清华大学出版社于 2007 年出版。此翻译版的出版和销售得到出版权和销售权的所有者 —— O'Reilly Media, Inc. 的许可。

版权所有，未经书面许可，本书的任何部分和全部不得以任何形式复制。

北京市版权局著作权合同登记
图字：01-2006-7115 号

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989 13501256678 13801310933

图书在版编目（CIP）数据

C++ Cookbook™ 中文版 /（美）斯蒂芬斯（Stepbens, D. R.）等著；金名等译．—北京：清华大学出版社，2007. 10
书名原文：C++ Cookbook
ISBN 978-7-302-15555-3

I. C… II. ①斯… ②金… III. C 语言－程序设计 IV. TP312

中国版本图书馆 CIP 数据核字（2007）第 098362 号

责任编辑：龙啟铭
封面设计：Karen Montgomery，张 健
责任校对：张 剑
责任印制：何 芊
出版发行：清华大学出版社 **地 址**：北京清华大学学研大厦 A 座
http://www.tup.com.cn **邮 编**：100084
c－service@tup.tsinghua.edu.cn
社 总 机：010-62770175 **邮购热线**：010-62786544
投稿咨询：010-62772015 **客户服务**：010-62776969
印 刷 者：北京市世界知识印刷厂
装 订 者：北京市密云县京文制本装订厂
经 销：全国新华书店
开 本：178 毫米×233 毫米 31.25 印张 字数：678 千字
版 次：2007 年 10 月第 1 版 **印 次**：2007 年 10 月第1 次印刷
印 数：1～3000 册
定 价：59.00 元（册）

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。联系电话：（010）62770177 转 3103 产品编号：023389－01

www.TopSage.com

O'Reilly Media, Inc. 介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构O'Reilly Media, Inc. 授权清华大学出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在 Unix、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的 *The Whole Internet User's Guide & Catalog*（被纽约公共图书馆评为20世纪最重要的50本书之一）到GNN（最早的Internet门户和商业网站），再到WebSite（第一个桌面PC的Web服务器软件），O'Reilly Media, Inc. 一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc. 是最稳定的计算机图书出版商 —— 每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得 O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc. 还有许多固定的作者群体 —— 他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc. 依靠他们及时地推出图书。因为 O'Reilly Media, Inc. 紧密地与计算机业界联系着，所以 O'Reilly Media, Inc. 知道市场上真正需要什么图书。

www.TopSage.com

译者序

C++是从C语言发展而来的。为了克服C语言存在的缺点，贝尔实验室的Bjarne Stroustrup博士及其同事开始对C语言进行改进和扩充，将“类”的概念引入了C语言，构成了最早的C++语言（1983年）。后来，Stroustrup和他的同事们又为C++引进了运算符重载、引用、虚函数等许多特性，并使之更加精炼，于1989年推出了AT & T C++ 2.0版。随后美国国家标准协会ANSI（American National Standard Institute）和国际标准化组织ISO（International Standards Organization）一起进行了标准化工作，并于1998年正式发布了C++语言的国际标准ISO/IEC:98-14882。C++支持面向对象的程序设计方法，特别适合于中型和大型的软件开发项目，从开发时间、费用到软件的重用性、可扩充性、可维护性和可靠性等方面，C++均具有很大的优越性。同时，C++又是C语言的一个超集，这就使得许多C代码不经修改就可被C++编译器编译通过。

C++基本上可以运行在所有的平台上，并且有无数的应用程序都是用C++实现的。无论你正在实现什么、你的目标平台是什么，你都可能碰到其他的研发人员已经解决很多年的那些共同的问题。本书就是用来解决这些共同的问题，并对每一个解决方案进行了解释。

本书的结构非常简单明了，基本上每一节都是由一个问题描述和一个代码解决方案组成，并且紧跟着是一个讨论。从本书中，你一定能找到那些正让你束手无策的问题的解决方案。

本书由周成兴、金名、张长富主译，黄中敏、马静静、冯华君、宋明钧、刘守燕、杨咏梅、魏敬安、朱建波、徐志平、赵杰辉、傅祎、郭碧莲、郭洵、洪晓煜、黄宜达、江松波、柯渝、赖曲芳、廖阳、刘文红、贺军、王雷、戴军、罗友平、邓琳、王益、廖雨果、禹莹、陈河南等人也参与了本书的部分翻译工作。

www.TopSage.com

www.TopSage.com

大家网
TopSage.com

目录

前言 1

第 1 章 创建 C++ 应用程序 7

1.0 概述 7
1.1 获得和安装 GCC 19
1.2 从命令行创建 "Hello, World" 应用程序 21
1.3 从命令行创建静态库 26
1.4 从命令行创建动态库 28
1.5 从命令行创建复杂的应用程序 34
1.6 安装 Boost.Build 40
1.7 使用 Boost.Build 创建 "Hello, World" 应用程序 42
1.8 使用 Boost.Build 创建静态库 45
1.9 使用 Boost.Build 创建动态库 46
1.10 使用 Boost.Build 创建复杂应用程序 47
1.11 使用 IDE 创建静态库 49
1.12 使用 IDE 创建动态库 52
1.13 使用 IDE 创建复杂应用程序 55
1.14 获得 GNU make 60

1.15 使用 GNU make 创建 "Hello, World" 应用程序 62
1.16 使用 GNU make 创建静态库 68
1.17 使用 GNU make 创建动态库 72
1.18 使用 GNU make 创建复杂应用程序 74
1.19 定义宏 77
1.20 从 IDE 指定命令行选项 79
1.21 生成调试版程序 80
1.22 生成发布版程序 83
1.23 指定运行库 86
1.24 严格遵循 C++ 标准 89
1.25 使源文件自动链接到指定的库 92
1.26 使用输出模板 94

第 2 章 代码结构 97

2.0 概述 97
2.1 确保头文件只包含了一次 98
2.2 保证一个跨多个源文件的变量只有一个实例存在 100
2.3 使用向前的类声明来减少 #include 的使用 101
2.4 使用命名空间来阻止名字冲突 103
2.5 包含一个内联文件 108

第 3 章 数值计算 110

3.0 概述 110
3.1 将字符串转换为数值 110
3.2 将数值转换为字符串 112
3.3 测试字符串中是否含有有效的数值 115
3.4 在限定精度内进行浮点数的比较 117
3.5 解析含有用科学计数法表示的数值的字符串 119
3.6 不同数值类型之间的转换 121
3.7 获得某种数值类型的最小和最大值 123

www.TopSage.com

第 4 章 字符串与文本 126

4.0 概述 126
4.1 填充字符串 127
4.2 裁剪字符串 128
4.3 将字符串存储到字符系列中 133
4.4 获得字符串的长度 136
4.5 翻转字符串 138
4.6 分割字符串 139
4.7 标记字符串 141
4.8 连接字符串 143
4.9 在字符串中进行查找 145
4.10 查找第 n 个子字符串 148
4.11 从字符串中删除一个子字符串 150
4.12 将字符串转换为小写或大写 151
4.13 大小写不敏感的字符串比较 154
4.14 大小写不敏感的字符串查找 156
4.15 在文本文件中进行制表符与空格的相互转换 157
4.16 文本文件中的换行 159
4.17 计算文本文件中的字符数、字数和行数 161
4.18 计算文本文件中每个字的出现次数 163
4.19 给文本文件添加空白 166
4.20 调整文本文件 168
4.21 将文本文件中的多个空格压缩成一个 170
4.22 自动改正文本 171
4.23 读取由逗号分隔开的文本文件 173
4.24 使用正则表达式来分割字符串 175

第 5 章 日期与时间 177

5.0 概述 177
5.1 获得当前日期与时间 177

www.TopSage.com

5.2 将日期/时间格式化为一个字符串 179
5.3 日期与时间运算 182
5.4 在时区之间转换 183
5.5 确定一年中的某天 184
5.6 定义受限数据类型 186

第 6 章 数据管理 190

6.0 概述 190
6.1 使用矢量而不是数组 191
6.2 高效地使用矢量 194
6.3 矢量复制 198
6.4 将指针存储在矢量中 200
6.5 将对象存储在链表中 201
6.6 将字符串映射到其他事物 206
6.7 使用哈希函数 212
6.8 按顺序存储对象 216
6.9 容器存储 219

第 7 章 算法 222

7.0 概述 222
7.1 遍历容器 223
7.2 删除对象 229
7.3 随机打乱数据 232
7.4 比较元素 233
7.5 合并数据 236
7.6 排列元素 240
7.7 分割元素 242
7.8 在序列上进行集合运算 244
7.9 序列元素转换 247
7.10 编写自己的算法 249
7.11 把元素输出到数据流 251

www.TopSage.com

第8章 类 255

8.0 概述 255
8.1 初始化类成员变量 256
8.2 使用函数创建对象 258
8.3 使用构造函数和析构函数管理资源 260
8.4 自动添加新的类实例 262
8.5 确保一个成员变量只有一个副本 264
8.6 在运行时确定对象的类型 265
8.7 确定某个对象的类是否是另一个类的子类 267
8.8 为类的每个实例指定一个唯一的标识符 269
8.9 创建单个类 271
8.10 创建含抽象基类的接口 273
8.11 编写一个类模板 277
8.12 编写一个成员函数模板 281
8.13 重载自增和自减运算符 283
8.14 为直观类行为重载算术和赋值操作符 286
8.15 调用一个超类虚函数 292

第9章 异常与安全性 294

9.0 概述 294
9.1 创建一个异常类 294
9.2 创建一个异常安全的构造函数 298
9.3 创建一个异常安全的初始化列表 301
9.4 创建异常安全的成员函数 304
9.5 安全地复制一个对象 308

第10章 数据流与文件 312

10.0 概述 312
10.1 文本输出对齐 313
10.2 格式化浮点数输出 316

www.TopSage.com

大家网 TopSage.com

10.3 编写自己的流操作程序 319
10.4 使类可写入流 323
10.5 使流可读取类 325
10.6 获取文件信息 327
10.7 文件复制 329
10.8 删除或重命名文件 332
10.9 创建一个临时文件名和文件 333
10.10 创建目录 335
10.11 删除目录 337
10.12 读取目录的内容 340
10.13 从字符串中提取文件扩展名 341
10.14 从完整路径中提取文件名 342
10.15 从完整路径和文件名中提取路径 344
10.16 替换文件扩展名 345
10.17 将两个路径组合成一个路径 346

第 11 章 科学与数学计算 349

11.0 概述 349
11.1 计算一个容器中元素的个数 350
11.2 查找最大或最小值 351
11.3 计算总和与平均值 353
11.4 过滤掉给定范围之外的值 356
11.5 方差、标准差及其他统计函数计算 357
11.6 生成随机数 359
11.7 使用随机数初始化容器 361
11.8 表示一个数目可变的数字矢量 362
11.9 表示一个数目固定的数字矢量 364
11.10 点积计算 366
11.11 计算矢量的范数 367
11.12 计算两个矢量之间的距离 368

www.TopSage.com

11.13 实现一个跨步迭代器 370
11.14 实现一个大小可变的矩阵 373
11.15 实现一个大小固定的矩阵 375
11.16 矩阵相乘 378
11.17 计算快速傅里叶转换 379
11.18 使用极坐标 381
11.19 在位集上进行算术运算 382
11.20 表示固定宽度的大整数 386
11.21 实现固定小数位的数值 389

第 12 章 多线程 391

12.0 概述 391
12.1 创建一个线程 392
12.2 使资源是线程安全的 395
12.3 从一个线程中给另一个线程发送通知 402
12.4 只初始化一次共享资源 405
12.5 给线程函数传递一个参数 406

第 13 章 国际化问题 409

13.0 概述 409
13.1 硬编码 Unicode 字符串 410
13.2 读取数字 411
13.3 读取日期和时间 414
13.4 读取货币值 418
13.5 给本地化字符串排序 422

第 14 章 XML 425

14.0 概述 425
14.1 解析一个简单的 XML 文档 426
14.2 使用 Xerces 字符串 433

www.TopSage.com

14.3 解析一个复杂的 XML 文档 435
14.4 XML 文档操作 444
14.5 使用 DTD 验证一个 XML 文档 448
14.6 使用 Schema 验证一个 XML 文档 452
14.7 使用 XSLT 转换一个 XML 文档 455
14.8 评估一个 XPath 表达式 460
14.9 使用 XML 保存和还原对象集 466

第 15 章 杂项 471

15.0 概述 471
15.1 为回调使用函数指针 471
15.2 使用指向类成员的指针 473
15.3 确保函数不会修改参数 475
15.4 确保成员函数不会修改其对象 477
15.5 编写一个不是成员函数的操作符 479
15.6 初始化由逗号分隔开的数列 481

www.TopSage.com

前言

C++基本上可以运行在所有的平台上，并且有无数的应用程序都是用C++实现的。如果你买了或者可能买这本书，你很有可能就是一个实现这些程序之一的工程师或者研究人员。但是无论你正在实现什么、你的目标平台是什么，你都可能碰到其他的研发人员已经解决很多年的那些共同的问题。本书就是用来解决这些共同的问题并对每一个解决方案进行了解释。

无论你是一个对C++非常熟悉、具有很多年经验的人员还是刚接触这门语言的工程师，你都可能非常熟悉在每一个项目中都需要重新实现的那些功能：日期和时间解析/算术、操作字符串和文本、操作文件、解析XML文档、使用标准容器等等。对这些问题，本书都包含它们的一些解决方案。在某些情况下（例如，时间和日期算术），标准类库基本上没有提供什么支持。而在其他一些情况下（例如，字符串操作）标准类库则包含很丰富的功能支持，但是它们也不是万能的，同样还是有很多共同的问题非常的麻烦。

每节的格式都是直截了当的。基本上每一节都一个问题描述和一个代码解决方案，并且紧跟着是一个讨论。每一节的讨论都是尽量地简短和适用，尽量做到把问题解决清楚但又不啰嗦，但是还是有一些有用的相关的主题，我们用了较长的篇幅来讨论它们。

本书是用来解决C++中的一些共同问题的，而不是C++的一本入门书籍。在讨论前，我们都假设你最少对C++和面向对象编程有一些基本的了解。特别地，如果你熟悉以下这些技术会对你有所帮助：

- C++的继承和虚函数
- 标准类库

www.TopSage.com

- 标准模板库中的一些组件（容器、迭代器和算法等）
- 模板

当然当你读这本书时这些也不是必要的，但是最少有些了解则对你理解本书的内容会有所帮助。

关于例子程序

在设计我们的例子代码时，我们尽量做到简单、可移植和比较好的性能。每一个解决方案的设计都追求相同的设计思路：有可能就使用标准C++（语言或库）；如果不能这样，就使用事实上的标准作为替代。例如，许多的章节都使用标准的字符串类来处理字符串，并且大部分数学和科学计算章节使用标准的数字类型、容器和模板。标准的类库在这些领域都有很强的支持，因此标准类库的使用是非常合适的。但是，相反在多线程或者XML解析中C++则支持很差。因此，我们就使用Boost线程库来处理多线程问题，用Xerces解析器来处理XML的解析问题。

通常，在C++中，对同一个问题有很多种解决办法，这给开发人员提供了灵活性，但是这也带来了一些争论。大部分例子都是我们能提供的最好的解决方案，但是这并不意味着这永远就是一个最好的方案。如果在某些方面存在更好的替代方案或者比较起来更好的办法（或许使用标准类库的方案是笨拙的、没创意的；这种情况下，我们使用了Boost库来提供一种可替代的方案），我们将展示某些可能的替代方案从而给你打开一些思路。

大部分例子都使用模板。如果你对于使用模板没有什么经验，你还是应尽快地学习一些这方面的知识。在这本书中对于模板没有特别多的介绍，除了在第8章中的8.11节和8.12节有些介绍之外没有更多的介绍。用C++开发的大部分兴趣出现在模板的元程序设计和基于策略的设计这些领域。

谈到C++的文献资料，在C++社区中有很多的跟踪。第一个技术报告（称为TR1）多少还有点不稳定。它列出了一些在C++的下一个版本中将会加进去的特性。标准类库并不要求支持它，但是很多的厂商已经开始实现TR1并且在不久的将来你可以看到有些特性会出现在它们的编译器中。TR1中的大部分库都出现在了Boost项目中。

我们使用Boost库更多一些。Boost是一个开源的、可移植的类库，它填补了标准类库中很多的缺陷。目前的版本是1.32，1.33版也快完成了。在一些例子中我们提供了许多指向特定Boost库的指针。更多的关于Boost库的信息请参考网站www.boost.org。

本书中使用到的一些惯例

下面的一些字体格式是本书使用的一些惯例：

www.TopSage.com

大家网 TopSage.com

斜体（*Italic*）

表示新的术语、URL、电子邮件、文件名、文件扩展名、路径名、目录、Unix实用程序、命令和命令行参数。

<...>

尖括号是用来包含这些元素，当这些元素以斜体格式出现时，这些元素是你在命令和命令行中需要指明的。

等宽字体（Constant Width）

指明其中的代码或片断。例如，类名、方法名、以及无论它们什么时候出现时呈现的常数宽度。

等宽粗体（**Constant Width Bold**）

表明用户在混合输入/输出例子中的输入。

等宽斜体（*Constant Width Italic*）

表明在语法例子中用户指定的项目。

使用例子代码

本书旨在帮你更好地完成工作。总体来说，你可以在你的程序和文档中使用本书中的代码。你不用与我们联系来得到允许，除非你打算复制这些代码的大部分。例如，使用本书中的很多代码来实现一个程序不需要得到允许。但是出售或者发布来自O'Reilly书中的例子的CD-ROM则需要得到我们的允许。引用本书和本书中的例子来回答问题不需要得到我们的允许。但把本书中大量的例子代码合并到你的产品文档中则需要得到我们的允许。

我们会很感谢你指明出处，但也不是必要的。一个出处常常包含标题、作者、出版商和ISBN号。例如："*C++ Cookbook* by D. Ryan Stephens, Christopher Diggins, Jonathan Turkanis, and Jeff Cogswell. Copyright 2006 O'Reilly Media, Inc., 0-596-00761-2。"

基于以上的授权，如果你在使用我们的例子代码时需要得到我们的允许，请与我们联系：*permissions@oreilly.com*。

意见和问题

有关这本书的意见和问题，请致信以下出版商：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

www.TopSage.com

中国：

100080 北京市海淀区知春路 49 号希格玛公寓 B 座 809 室
奥莱理软件（北京）有限公司

本书的 Web 页上列出了勘误表、示例和任何额外的信息。可登录以下网址查询：

http://www.oreilly.com/catalog/cpluspluscbk
http://www.oreilly.com.cn/book.php?bn=978-7-302-15555-3

如果想就本书的技术问题发表评论或咨询，请发邮件至：

bookquestions@oreilly.com
info@mail.oreilly.com.cn

关于本书和其他书籍的更多信息，请访问 O'Reilly 的网站：

http://www.oreilly.com
http://www.oreilly.com.cn

致谢

来自 D. Ryan Stephens

我想感谢的最重要的人就是我的妻子 Daphne 和我的孩子 Jesse、Pascal 和 Chloe。完成这本书的写作是困难的，但是最主要的是它是一项很费时间的工作，而我的家庭大力支持并且以最好的方式容忍我深夜还在办公室工作。

我还需要感谢技术审查人员，正是他们使得这本书的质量如此好。与其他的很多事情一样，正是他们一遍遍的检查使得本书的清晰性和准确性得以保证。非常感谢 Dan Saks、Uwe Schnitker 和 David Theese。

最后，感谢我的编辑 Jonathan Gennick， 正是因为他的建议，使得本书的语法、格式以及我的写作心态都一直很好。

来自 Christopher Diggins

感谢 Kris Unger、Jonathan Turkanis、Jonathan Gennick 和 Ryan Stephens，感谢他们给我带来帮助的建议和批评，使得我能成为一个更好的作者。特别要感谢我的妻子 Meanie Charbonneau，因为她照亮了我的生活。

www.TopSage.com

来自 Jonathan Turkanis

由于我负责的章节涉及到非常多的不同商业产品和开源项目，并且又由于我对这些项目和产品又有很多的问题，所以我需要感谢很多的人。

首先，让我感谢Ron Liechty、Howard Hinnant和Metrowerks的工程师，因为他们回答了我的每一个可能的问题并且给我提供了CodeWarrior的好几个版本。

我还要感谢Boost.Build的开发人员，特别是Vladimir Prus、Rene Rivera和David Abrahams，不仅仅是因为他们回答了我的问题，而且还因为他们集成了Boost build系统，这对于第1章中的信息提供了一个唯一的非常重要的来源。

同样，还要感谢Digital Mars的Walter Bright；Comeau Computing的Greg Comeau；Dinkumware的P. J. Plauger；Bloodshed Software的Colin Laplace； borland.public.* newsgroups的Ed Mulroy和Pavel Vozenilek；microsoft.public.vc.languages的Arnaud Debaene和Igor Tandetnik；MinGW/MSYS邮件列表中的Earnie Boyd、Greg Chicares、Adib Taraben、John Vandenberg和Lennart Borgman；Cygwin列表中的Christopher Faylor、Larry Hall、Igor Pechtchanski、Joshua Daniel Franklin和Dave Korn；GCC开发者列表中的Mike Stump和Geoffrey Keating；DecisionSoft的Mark Goodhand；以及apache.org的David N. Bertoni。

我还要感谢Robert Mecklenburg，因为他的书*Managing Projects with GNU make*，Third Edition (O'Reilly)给我提供了很多GNU make工具的基础知识。

此外，我还要感谢Vladimir Prus、Matthew Wilson、Ryan Stephens和Christopher Diggins，正是他们提供了我的前期手稿的批评意见。

最后，我还得感谢我的编辑Jonathan Gennick、我的妻子Jennifer和我的祖父Louis S. Goodman，正是他教会了我如何写作。

www.TopSage.com

www.TopSage.com

第1章

创建C++应用程序

1.0 概述

本章介绍了把C++源代码转换成可执行程序和库的一些技巧。通过这些技巧的学习，你将学会那些用于构建C++应用程序的基本工具、在构建过程中包含的各种二进制文件，以及用于构建C++应用程序的系统。

如果你看看本章中的各种技巧，你可能会有这种印象："我已经反复解决过这些问题了"。这就对了。这是因为构建C++应用程序的方法有很多，而我不可能介绍全部，我只是介绍了一些最重要的方法。在前面十几个技巧中，介绍了用各种不同方法来完成三种基本任务：构建静态库、构建动态库以及构建可执行程序。这些按使用的技巧进行分组：首先是从命令行构建，接着是用Boost构建系统（Boost.Builde），然后是用集成开发环境(IDE)，最后是用GNU *make*。

在开始阅读本书介绍的技巧之前，应先阅读一下后面一些介绍性的章节。这里我将解释一些基本的术语，对本章要介绍的命令行工具、构建系统和IDE给出一个概述，并引入一些源代码示例。

注意： 即使你将使用的是构建系统或IDE，我还是建议你先阅读一下从命令行构建应用程序的技巧：这些技巧介绍了一些基本概念。你在学习本章后面的内容时将用到这些概念。

基本术语

用于构建C++应用程序的三个基本工具是编译器、链接器和归档库。这些程序和其他一些可能用到的工具集合就称为工具集。

编译器以C++源文件为输入，生成目标文件（*object file*）。目标文件包含了机器可执行代码、数据和函数的符号引用。归档器（*archiver*）以目标文件为输入，生成静态库（*static*

www.TopSage.com

大家网 TopSage.com

library)。链接器以目标文件集和库为输入，将符号引用解析，生成可执行程序或动态库。简单地说，链接器是把每个符号与其定义相匹配。当创建好了可执行程序或动态库后，我们就说已链接好了，用于构建可执行程序或动态库的库文件就称为已被链接了。

一个可执行程序（或称为应用程序）就是所有可被操作系统运行的程序。动态库（也称为共享库）与可执行程序类似，只不过它不能自己运行，它由机器可运行代码体组成，当应用程序启动后，动态库的代码体被加载到内存中，且可以被一个或多个应用程序共享。在 Windows 操作系统中，动态库又称为动态链接库（*dynamic link libraries*, DLL）。

只有在构建可执行程序时，才需要该可执行程序所需的目标文件和静态库。但是，当可执行程序运行时，它所依赖的动态库必须在用户的系统中。

表 1-1 显示了在 Microsoft Windows 和 Unix 操作系统中，以上四种基本文件类型的文件扩展名。当我说 Windows 和 Unix 下的某个扩展名不同的文件时，如果从内容上能加以区分，经常会省略其扩展名。

表 1-1 Windows 和 Unix 操作系统下的文件扩展名

文件类型	Windows	Mac OS X	Unix
目标文件	*.obj*	*.o*	*.o*
静态库	*.lib*	*.a*	*.a*
动态库	*.dll*	*.dylib*	*.so*
可执行程序	*.exe*	无扩展名	无扩展名

注意： 在本章中，如果提到 Unix，则同样适用于 Linux。

注意： 当你构建本章中的示例时，你的工具可能会生成很多带扩展名的辅助文件，且这些扩展名没有出现在表 1-1 中。除非特别说明，否则你可以安全地忽略这些文件。如果你真的想知道这些文件是干什么用的，请查阅你的工具集文档说明。

IDE 与构建系统

编译器、链接器和归档器都是命令行工具，它们都是从外壳程序下运行的。例如，在 Unix 系统下的外壳是 *bash*，在 Windows 系统下是 *cmd.exe*。输入文件名和输出文件名与其他所需的配置信息一起在命令行中作为文本传递给编译器、链接器和归档器。但是，手工使用这些工具很繁琐。即使是一个小项目，要记住每个工具的命令行选项以及项目源文件和二进制文件编译和链接的顺序，是一件很困难的事。当源文件做了修改时，你就应确定哪些目标文件需要重新链接。如果你重建的文件比实际需要的更多，那么就浪费时间了；如果重建的文件不够多，那么你就会失败或得到的是含有 bug 的应用程序。对于

www.TopSage.com

含有几千个独立文件（如源文件、目标文件、库以及可执行程序）的大型C++项目，用命令行来构建简直是不可能的事情。

构建大型C++应用程序有两个基本的方法：

- IDE提供了一个图形界面，用于组织源文件集合，说明要从中生成哪些二进制文件。一旦指定好了这些信息，就可以从菜单或工具栏中选取相应的命令来生成这些二进制文件。IDE负责确定二进制文件的生成顺序、生成它们需要哪些工具，以及应给这些工具传递哪些命令选项。无论你修改了一个或多个源文件，你都可以让IDE只重新生成那些已过时的二进制文件。

 IDE把源文件组织成集合，称为项目。一个IDE项目通常只与某一个二进制文件或一个二进制文件的不同形式关联，如应用程序的调试或发布版本。大多数IDE还允许用户把项目组织成项目组或解决方案，并在组中指定项目之间的相互关系。

- 构建系统提供一个文本文件格式，用于描述源文件集合，以及从源文件中要生成的二进制文件，包括一个构建工具，它可以读取这些文本文件，并通过调用适当的命令行工具来生成二进制文件。通常，这些文本文件是利用文本编辑器来生成和编辑的，而构建工具则是从命令行来调用的。但是，一个构建系统提供了一个图形界面，用于编辑文本和调用构建工具。

 IDE是把文件组织成项目，而构建系统则是把文件组织成目标。大多数目标都对应于必须生成的二进制文件，而其他的目标则对应于构建工具必须执行的工作，例如，安装一个应用程序。

最常见的构建工具是*make*实用程序，它所依赖的文本文件称为*makefile*文件。*Make*实用工具有很多版本。本章将要介绍的是GNU的*make*实用工具，它功能最强大，有很好的可移植性。GNU *make*是一个非常灵活的工具，它不仅可用来构建C++应用程序。它还具有被开发人员广泛使用且很好掌握的优点。但是，要让GNU *make*实用程序完全按你所想的那样工作是一件比较困难的事，对于含有多个工具集的复杂项目更是如此。由于这种原因，我还将介绍*Boost.Build*，它是一个功能强大、可扩展的构建系统，自始至终都是为构建C++应用程序而设计的。

注意：关于GNU *make*的完整介绍，请参阅*Managing Projects with GNU make*, Third Edition (Robert Mecklenburg, O'Reilly)。

Boost.Build是由Boost C++库项目的成员开发的。它已被C++社区开发人员使用多年了，现在仍在积极开发之中。Boost.Build使用一个名为*bjam*的构建工具和名为*Jamfiles*的文本文件编辑器。其最大的功能是，你可以轻松地管理包含多平台和多种构建配置的复杂项目。尽管Boost.Build开始时是作为Perforce的*Jam*构建系统的扩展，但它经过了大面

www.TopSage.com

积的重新设计。就在出版本书时，Boost.Build开发人员正准备发布该构建系统第二个主要版本的正式版，本章介绍的就是这个版本。

工具集概述

本章将介绍7种命令行工具集：GCC、Visual C++、Intel、Metrowerks、Borland、Comeau和Digital Mars。表1-2显示了不同工具集的命令行工具名。表1-3显示了安装这些工具集后它们在系统中的位置。Windows的工具名按Windows可运行文件的要求使用了*.exe*后缀名，对于Windows和Unix系统都可用的工具集，则把后缀放在括号中了。

表1-2 不同工具集的命令行工具名

工具集	编译器	链接器	归档器
GCC	*g++[.exe]*	*g++*	*ar[.exe]* *ranlib[.exe]*
Visual C++	*cl.exe*	*link.exe*	*lib.exe*
Intel (Windows)	*icl.exe*	*xilink.exe*	*xilib.exe*
Intel (Linux)	*icpc*	*icpc*	*Ar.* *ranlib*
Metrowerks	*mwcc[.exe]*	*mwld[.exe]*	*mwld[.exe]*
Comeau	*como[.exe]*	*como[.exe]*	与具体的工具集有关
Borland	*bcc32.exe*	*bcc32.exe ilink32.exe*	*tlib.exe*
Digital Mars	*dmc.exe*	*link.exe*	*lib.exe*

表1-3 命令行工具的位置

工具集	位置
GCC (Unix)	通常为*/usr/bin*或*/usr/local/bin*
GCC (Cygwin)	Cygwin安装位置下的*bin*子目录
GCC (MinGW)	MinGW安装位置下的*bin*子目录
Visual C++	Visual Studio安装位置下的*VC/bin*子目录
Intel (Windows)	Intel编译器安装位置下的*bin*子目录
Intel (Linux)	Intel编译器安装位置下的*bin*子目录
Metrowerks	CodeWarrior安装位置下的其他*Metrowerks*工具/命令行的*Tools*子目录
Comeau	Comeau安装位置下的*bin*子目录
Borland	C++Builder、C++BuilderX或Borland命令行安装位置下的*bin*子目录

www.TopSage.com

不要被这么多的工具集吓住你了，你没有必要全部学习它们。在大多数情况下，你完全可以忽略那些与你所用的工具集无关的内容。但是，如果你想了解其他一些工具集，就应阅读关于Visual C++和GCC的章节，因为这是在Windows和Unix系统下占主导地位的工具集。

下面让我们来看看这7种工具集。

GNU编译器集(GCC)

GCC是一个编译器集，可用于多种语言，包括C和C++。它因为是开源软件而出名的，几乎可以运行在每种平台上，而且高度遵循C++语言标准。在很多Unix平台上，它是占主导地位的编译器，在Microsoft Windows中也被广泛使用。即使GCC不是你主要使用的工具集，通过用GCC编译代码，你也可以学到很多东西。而且，如果你想以某种方式来提高你的C++语言，可以使用GCC代码来测试一下。

GCC有一个libstdc++库，这是一个C++标准库的开源实现。它可以与开源STLPort C++标准库和Dinkumware标准库一起使用。

注意：要获得GCC，请参见1.1节。

注意：本章中的GCC代码示例在GNU/Linux（Fedora Core 3）系统下用GCC 3.4.3和GCC 4.0.0进行了测试，在Mac OS X (Darwin 8.2.0)系统下用GCC 4.0.0进行了测试，在Windows 2000 Professional下用GCC 3.4.2（MinGW）和3.4.4（Cygwin）进行了测试。

Visual C++

在Windows平台上，Microsoft的工具集Visual C++是占主导地位的。尽管有多个旧版本仍在广泛使用，但最新版本则是高度遵循C++标准的。它还可以生成高度优化的代码。Microsoft的工具集是随着Visual C++和Visual Studio开发环境发布的，将在下一节中介绍。在写作本书时，它们还是Visual C++ Toolkit 2003的一部分，可以免费从*www.microsoft.com*下载。

Visual C++带有一个Dinkumware C++标准库实现的定制版本。Dinkumware的C++标准库是最高效、严格遵守商业实现的C++标准库。它可用于多种平台，包含有本章所介绍的其他工具集。

注意：本章的Visual C++示例是用Microsoft Visual Studio .NET 2003和Microsoft Visual Studio 2005 (Beta 2)测试的，参见表1-4。

www.TopSage.com

表1-4 Microsoft Visual Studio的不同版本

产品名	IDE版本	编译器版本
Microsoft Visual Studio	6.0	1200
Microsoft Visual Studio .NET	7.0	1300
Microsoft Visual Studio .NET 2003	7.1	1310
Microsoft Visual Studio 2005 (Beta 2)	8.0	1400

Intel

Intel开发了多种用于Intel处理器的C++编译器。它们以能生成非常快的代码著称，在Intel体系结构下可能是最快的了。它们基于来自Edison Design Group (EDG)的C++前端，它们也是高度遵循标准的。

Windows系统下的Intel C++编译器可以利用Microsoft的Visual C++或Visual Studio开发环境，要使Intel编译器很好地工作，必须安装这些开发环境。该编译器设计为与Visual C++相互兼容，它可以作为Visual C++开发环境的一个插件，生成的代码可以与Visual C++编译器所生成的代码兼容，它还提供了与Visual C++编译器同样多的命令行选项，甚至有与Microsoft调试器相媲美的功能。Intel C++ Compiler for Windows的商业版本可以从*www.intel.com*购买，还有价格合理的教育版。

与Intel的Windows系统下的C++编译器和Visual C++编译器相兼容一样，Intel的Linux系统下的C++编译器与GCC也是相互兼容的。它要求GCC来操作、支持很多GCC命令选项，默认地实现一些GCC语言扩展。Intel C++ Compiler for Linux的商业版本可以从*www.intel.com*购买。也可以免费下载非商业版本。

在Windows系统下，Intel编译器使用的标准库与Visual C++所带的相同，都是Dinkumware标准库，而在Linux系统下，它所使用的是libstdc++标准库。

注意： 本章的Intel示例是用GNU/Linux (Fedora Core 3)系统下的Intel C++ Compiler 9.0 for Linux测试的，以及用Windows 2000 Professional系统下的Intel C++ Compiler 9.0 for Windows测试的。

Metrowerks

Metrowerks的命令行工具是随CodeWarrior开发环境发布的，是最好用的，因为它不仅很好地遵循了C++标准，而且所生成代码的效率高。它们都带有MSL。MSL是Metrowerks对C++标准库的很好实现。到目前为止，Metrowerks为Windows、Mac OS和多种嵌入式平台开发了工具。但是，在2004年，Metrowerks把它的Intel x86编译器和调试器卖给了Nokia公司，并中止了Windows系统下的CodeWarrior产品线。2005年，在Apple

www.TopSage.com

大家网 TopSage.com

公司宣布计划使用Intel产品的芯片后，Metrowerks公司透露，即将推出的CodeWarrior 10 for Mac OS可能是该平台上的最后版本了。今后，Metrowerks公司将把精力集中在嵌入式开发上，目标是Freescale半导体公司生成的芯片。

注意： 当你阅读本书时，Metrowerks可能已经是Freescale半导体公司的一部分了，而Metrowerks也许不再与CodeWarrior产品线有关。但是，我仍将使用Metrowerks这个名字，因为还不明确未来的名字将会是什么。

注意： 本章的Metrowerks代码示例已经过Mac OS X (Darwin 8.2.0)系统下的CodeWarrior 9.6和10.0 (Beta)以及Windows 2000 Professional系统下的CodeWarrior 9.4测试。

Borland

Borland的命令行工具曾经被认为是相对不错的。但是，直到2005年9月，经历了3年才完成最新的主要升级，且只是对2000年的版本做了有限改进。因此，Borland的工具现在认为已相当过时了。2003年，Borland公司宣布对其使用EGD前端的C++编译器重新设计的计划，但在好长时间里，Borland都没有关于这个计划的新声明。Borland的命令行工具现在之所以重要，是因为它们仍在广泛使用。

当前，Borland命令行工具的最新版本可以通过购买C++ Builder（下一节介绍）或C++ BuilderX开发环境来获得，或者下载免费的C++ BuilderX个人版。

Borland工具集有两个C++标准：STLPort和Rogue Wave标准库的过时版。Borland还使用一个随Dinkumware标准库发布的工具。

注意： 本章的Borland示例经过Windows 2000 Professional上的Borland C++ Builder 6.0（编译器版本为5.6.4）测试。

Comeau

Comeau C++编译器被广泛地认为是最遵循标准的C++编译器。它除了实现最新版本的C++语言之外，还支持多种版本的C和大量早期的C++版本。它还是最便宜的，当前的价格是50美元。

与Intel编译器一样，Comeau使用的也是EDG前端，要求单独的C编译器才能正确工作。与Intel编译器不同的是，Comeau可以使用多种不同的C编译器作为其后端。

Comeau可用于Microsoft Windows和很多的Unix平台上。如果Comeau不能用于你的平台，你可以向Comeau Computing公司定购开发一个，但这样就更贵了。你可以从*www.comeaucomputing.com*定购Comeau编译器。

www.TopSage.com

注意：当我讨论Unix系统中的Comeau时，我都是假设其后端编译器为GCC。而当我讨论Windows系统中的Comeau时，我将根据后端的编译器指明命令行选项。但是，由于Comeau可以使用这么多的后端编译器，因而不太可能无遗漏。

Comeau带有libcomo库，该库是基于Silicon Graphics标准库的C++标准库实现的。它还可以使用Dinkumware'标准库。

注意：本章的Comeau示例假定你使用的是libcomo标准库。这些示例经GNU/Linux (Fedora Core 3)系统中的Comeau 4.3.3 and libcomo 31库测试，使用GCC 3.4.3作为后端编译器，以及经过Windows 2000 Professional系统中的Visual C++ .NET 2003测试。

Digital Mars

Digital Mars是由Walter Bright编写的C++编译器。你可以从*www.digitalmars.com*免费下载，也可以定购一个含有Digital Mars编译器、IDE和其他一些有用工具的CD。免费版的编译器可以用来编译本章中的所有Digital Mars示例代码，但那些要求运行时库的动态版本除外，这些运行时库只在CD上才有。

Digital Mars是一个非常快速的编译器，可以生成高度优化的代码。但现在在编译器使用高级模板的代码时会有一些问题。Walter Bright对这些bug的报告很积极，正在进行修改，使得Digital Mars遵循标准。

Digital Mars有两个标准库：STLPort标准库和老版本的标准库。为了向后兼容，STLPort必须显式地由用户启用。本章的所有Digital Mars代码示例都是使用STLPort标准库的。

注意：本章中的Digital Mars示例代码经Windows 2000 Professional系统平台上的Digital Mars 8.45测试过。

IDE概述

本章将介绍四种IDE：Microsoft Visual C++、Metrowerks CodeWarrior、Borland C++Builder和Bloodshed Software的Dev-C++。还有很多重要的IDE这里没有讨论，如Apple的Xcode和Eclipse Project就是这样突出的例子，但这里介绍的四种IDE将为你学习其他IDE提供一个良好的开端。

注意：与命令行工具一样，可以跳过你不关心的IDE介绍。

Visual C++

在Microsoft Windows系统中，Microsoft Visual C++是占主导地位的C++开发环境。

www.TopSage.com

它既可以作为一个单独的应用程序，也可以作为Visual Studio套装软件的一部分，它带有大量用于Windows应用开发的工具。作为一个可移植的C++开发环境，其最显著的特点有：

- 它是一个高度遵循C++标准的C++编译器。
- 它带有Dinkumware C++标准库。
- 它有一个很好的可视化调试器。
- 它有一个项目管理器，用于跟踪项目中的各个部分。

Visual Studio有多个版本在广泛使用中。由于不同版本的名字容易混淆，我在表1-4中列举了使用最广泛的几个版本。

CodeWarrior

CodeWarrior是Metrowerks的跨平台开发环境。它具有与Visual C++相同的特性，包括：

- 它是一个高度遵循C++标准的C++编译器。
- 它带有一个很好的C++标准库。
- 它有一个很好的可视化调试器。
- 它有一个项目管理器，用于跟踪项目中的各个部分。

CodeWarrior的优点之一是可用于多种平台之上，但是，其Windows产品线已经停工，而其Macintosh产品线将很快停工，但它为嵌入式开发保留了一个重要的平台。

注意：讨论CodeWarrior IDE时，假定你所使用的是CodeWarrior 10 for Mac OS X。其他平台的CodeWarrior IDE非常类似。

C++Builder

C++Builder是Borland公司用于Microsoft Windows应用程序开发的开发环境。它最吸引人的地方是支持Borland的Visual Component Library。但是，作为可移植的C++开发工具，它最主要的特性有：

- 它是一个成熟的C++编译器。
- 它含有STLPort标准库。
- 它有一个良好的可视化调试器。
- 项目管理器具有一定的处理不同项目的关联性的能力。

www.TopSage.com

我在这里介绍C++Builder是因为它被广泛使用，而且还有专注的用户群。

不要把C++Builder与C++BuilderX混淆了，后者是Borland公司于2003年发布的一个跨平台的开发环境。尽管C++BuilderX是一个很有用的开发工具，但它还没有取得商业成功，不能肯定Borland公司是否会发布更新版。

Dev-C++

Bloodshed Software公司的Dev-C++是移植用于Windows系统的免费的C++开发环境，它使用的是GCC的MinGW，具体见1.1节的介绍。它有一个很好用的文本编辑器，以及一个可视化GNU调试器界面。

Dev-C++为GCC的很多命令行选项提供了一个不完整的图形界面。在很多情况下，用户必须通过在文本框中输入命令行选项来配置他们的项目。而且，其项目管理器一次只能处理一个项目，其可视化调试器也不那么可靠。尽管有这些不足，Dev-C++还是有一群活跃的用户，包括很多大学生。对那些想学习C++但又不想购买任何C++开发工具的人来说，它是一个不错的开发环境。

```
John, Paul, George, and Ringo
```

自从Brian Kernighan和Dennis Ritchie于1978年出版了*The C Programming Language*以来，通常都是通过编写、编译和运行一个往控制台显示“Hello, World!”的程序来开始一种新语言的学习。由于本章要介绍静态和动态库，以及可执行程序，因此我将介绍一个稍微复杂一点的示例。

示例1-1～示例1-3显示了*hellobeatles*应用程序的源代码，该程序将往控制台显示：

John, Paul, George, and Ringo

该程序本来可以写成一个源文件，但我把它分成了三个模块：静态库*libjohnpaul*、动态库*libgeorgeringo*，以及可执行程序*hellobeatles*。而且，尽管每个库都可以很容易地实现为单个的头文件和*.cpp*文件，但我把实现分成了多个源文件，用于演示如何编译和链接含有多个源文件的项目。

注意：在开始本章的技巧之前，请先创建四个目录*johnpaul*、*geogreringo*、*hellobeatles*和*binaries*。前三个目录用于存放示例1-1～示例1-3的源文件。第四个用于存放由IDE生成的二进制文件。

*Libjohnpaul*的源代码如示例1-1所示。*Libjohnpaul*的公用接口由单个函数`johnpaul()`组成，它声明在头文件*johnpaul.hpp*中。函数`johnpaul()`负责向控制台显示：

```
John, Paul,
```

www.TopSage.com

johnpaul()的实现又分成两个源文件*john.cpp*和*paul.cpp*，每个源文件都只负责显示单个人名。

示例 1-1 libjohnpaul 的源文件

johnpaul/john.hpp

```
#ifndef JOHN_HPP_INCLUDED
#define JOHN_HPP_INCLUDED

void john( ); // Prints "John, "

#endif // JOHN_HPP_INCLUDED
```

johnpaul/john.cpp

```
#include <iostream>
#include "john.hpp"

void john( )
{
    std::cout << "John, ";
}
```

johnpaul/paul.hpp

```
#ifndef PAUL_HPP_INCLUDED
#define PAUL_HPP_INCLUDED

void paul( ); // Prints " Paul, "

#endif // PAUL_HPP_INCLUDED
```

johnpaul/paul.cpp

```
#include <iostream>
#include "paul.hpp"

void paul( )
{
    std::cout << "Paul, ";
}
```

johnpaul/johnpaul.hpp

```
#ifndef JOHNPAUL_HPP_INCLUDED
#define JOHNPAUL_HPP_INCLUDED

void johnpaul( ); // Prints "John, Paul, "

#endif // JOHNPAUL_HPP_INCLUDED
```

johnpaul/johnpaul.cpp

```
#include "john.hpp"
#include "paul.hpp"
#include "johnpaul.hpp"

void johnpaul( )
{
    john( );
    paul( );
}
```

libgeorgeringo 的源代码如示例 1-2 所示。*Libgeorgeringo* 的公用接口由单个函数

www.TopSage.com

大家网 TopSage.com

georgeringo()组成，它声明在*georgeringo.hpp*头文件中。正如你能猜测到的那样，georgeringo()函数负责向控制台显示：

```
George, and Ringo
```

同样，georgeringo()的实现也分成了两个源文件*george.cpp*和*ringo.cpp*。

示例 1-2 libgeorgeringo 的源文件

georgeringo/george.hpp

```
#ifndef GEORGE_HPP_INCLUDED
#define GEORGE_HPP_INCLUDED

void george( ); // Prints "George, "

#endif // GEORGE_HPP_INCLUDED
```

georgeringo/george.cpp

```
#include <iostream>
#include "george.hpp"

void george( )
{
    std::cout << "George, ";
}
```

georgeringo/ringo.hpp

```
#ifndef RINGO_HPP_INCLUDED
#define RINGO_HPP_INCLUDED

void ringo( ); // Prints "and Ringo\n"

#endif // RINGO_HPP_INCLUDED
```

georgeringo/ringo.cpp

```
#include <iostream>
#include "ringo.hpp"

void ringo( )
{
    std::cout << "and Ringo\n";
}
```

georgeringo/georgeringo.hpp

```
#ifndef GEORGERINGO_HPP_INCLUDED
#define GEORGERINGO_HPP_INCLUDED

// define GEORGERINGO_DLL when building libgerogreringo.dll
# if defined(_WIN32) && !defined(__GNUC__)
#  ifdef GEORGERINGO_DLL
#   define GEORGERINGO_DECL _  _declspec(dllexport)
#  else
#   define GEORGERINGO_DECL _  _declspec(dllimport)
#  endif
# endif // WIN32

#ifndef GEORGERINGO_DECL
# define GEORGERINGO_DECL
#endif
```

www.TopSage.com

```
// Prints "George, and Ringo\n"
#ifdef __MWERKS__
# pragma export on
#endif

GEORGERINGO_DECL void georgeringo(  );
#ifdef __MWERKS__
# pragma export off
#endif

#endif // GEORGERINGO_HPP_INCLUDED
```

georgeringo/ georgeringo.cpp

```
#include "george.hpp"
#include "ringo.hpp"
#include "georgeringo.hpp"

void georgeringo(  )
{
    george(  );
    ringo(  );
}
```

头文件 *georgeringo.hpp* 包含了一些复杂的预处理器指令。如果你现在不理解，没关系，1.4 节将解释它们。

最后，可执行程序*hellobeatles*的源代码如示例1-3所示。它由单个源文件*hellobeatles.cpp*组成，该源文件包含了头文件*johnpaul.hpp*和*georgeringo.hpp*，调用了函数`johnpaul()`和`georgeringo()`。

示例 1-3 hellobeatles 的源代码

hellobeatles/ hellobeatles.cpp

```
#include "johnpaul/johnpaul.hpp"
#include " georgeringo/ georgeringo.hpp"

int main(  )
{
    // Prints "John, Paul, George, and Ringo\n"
    johnpaul(  );
    georgeringo(  );
}
```

1.1 获得和安装 GCC

问题

如何获得 GCC，即免费的 GNU C/C++ 编译器？

解决方案

其解决方案取决于你的操作系统。

Windows

安装 MinGW、Cygwin，或者两者都安装。

www.TopSage.com

要安装MinGW，可以到MinGW的主页*www.mingw.org*上找到MinGW的下载页面，下载MinGW安装程序的最新版本，其名字应为*MinGW-<版本号>.exe*。

接下来就可以运行安装程序。它将询问你要把MinGW安装到何处。它还会询问你要安装哪些包。在最小安装的情况下，将安装*gcc-core*、*gcc-g++*、*binutils*和MinGW运行库。但你也可以安装更多的东西。安装完成后，你就可以从Windows的命令行运行*gcc*、*g++*、*ar*、*ranlib*、*dlltool*和其他几个GNU工具。你还可以把MinGW安装的*bin*子目录添加到环境变量PATH中，这样你就只需在命令行中输入这些工具的名字就可以运行它们，而不必输入完整的路径名。

要安装Cygwin，可以到Cygwin的主页*www.cygwin.com*上找到Cygwin的下载链接，下载Cygwin安装程序。接下来，运行安装程序，它将询问你一系列的选择，例如，要把Cygwin安装到何处。

注意：这里将详细介绍Cygwin的安装过程，因为根据你所安装的不同内容，其过程有一点复杂。当你阅读本书时，该过程可能会发生变化，倘若果真如此，也许是变得更容易了。

最为重要的是选择要安装哪些包。如果有足够的磁盘空间和高速的因特网连接，建议你安装所有包。如果磁盘空间不足，或者因特网连接速度慢，可以选择安装少一些的包。

当安装完成后，Cygwin安装目录中应含有一个名为*cygwin.bat*的文件。运行这个脚本文件将显示Cygwin外壳程序，这是一个命令行环境，从这里你可以运行*gcc*、*g++*、*ar*、*ranlib*、*dlltool*、*make*和其他已安装的实用工具。安装过程会把Cygwin的bin子目录添加到PATH环境变量中，这样就可以从Windows的外壳程序*cmd.exe*中运行这些实用工具了。但是，你将发现，Cygwin外壳程序（它是*bash*外壳程序的一部分）是运行GNU实用工具的友好环境。

Unix

从命令行输入**g++ -v**来检测在你的系统中是否已经安装了GCC。如果已经安装了GCC，且支持C++语言，则将显示以下信息：

```
Using built-in specs.
Target: powerpc-apple-darwin8
Configured with: /private/var/tmp/gcc/gcc-5026.obj~19/src/configure
--disable-checking --prefix=/usr ...
```

如果没有安装GCC，或者已安装但不支持C++，那么你就得自己安装了。根据你的平台的不同，这将是一个复杂的过程，你得安装GNU *make*和GNU *binutils*包。详细说明可参见*gcc.gnu.org/install*网站。

如果你使用的是Mac OS X操作系统，获得GCC最容易的方法是从Apple公司的网站上

www.TopSage.com

下载Xcode开发环境，然后遵循安装说明进行即可。现在，Xcode可以从*developer.apple.com/tools*下载。

如果使用的是Linux操作系统，则某些版本的GCC可能已经安装了，输入`g++ -v`可以确定其版本号。GCC的最新版本是4.0.0。如果你的版本不够新，可以使用Linux发布版的包管理系统来安装最新的版本。

讨论

Cygwin和MinGW代表了把GNU工具安装到Wdindows系统中的最困难的方法。Cygwin是一个目标远大的项目，可以生成Windows系统下类似于Unix的环境。它提供了一个与Unix兼容的层，允许为Unix编写的程序在Windows上编译和运行。因此，Cygwin有很多Unix的实用程序。即使你不是Unix开发人员，你也会很快就认为Cygwin工具是必不可少的。

MinGW表示的意义是“Minimalist GNU for Windows”，它提供了使用GCC创建Windows可执行程序的最小环境。另外，MinGW还包含有GCC、GNU归档器和链接器，以及GNU调试器GDB。它还包含有MSYS，这是一个命令行环境，用于运行GNU make文件和配置脚本。MSYS将在1.4节中介绍。

Cygwin和MinGW一个重要的不同是关于授权问题。除少数情况外，你可以发布使用MinGW编译的二进制文件。但是如果要发布用Cygwin编译的二进制文件，默认情况下要包含GNU General Public License (GPL)。如果你想发布由Cygwin编译的程序而不带源文件，你必须从Red Hat公司购买授权。详细信息请参见Cygwin和MinGW网站。

参见1.4节。

1.2 从命令行创建“Hello, World”应用程序

问题

如何创建如示例1-4所示的简单的“Hello, World”程序？

示例1-4 一个简单的“Hello, World”程序

hello.cpp

```
#include <iostream>

int main(  )
{
   std::cout << "Hello, World!\n";
}
```

解决方案

可按照以下步骤：

www.TopSage.com

1. 设置好工具集所要求的环境变量。
2. 输入一个命令，告诉编译器编译和链接程序。

用于设置环境变量的脚本程序如表1-5所示。这些脚本程序位于命令行工具的相同目录中(见表 1-3 所示)。如果你所用的工具集没有出现在表 1-5 中，那么你就可以跳过第 1 步。否则，如果你使用的是 Windows 系统，则运行相应的脚本程序；如果使用的是 Unix 系统，则编写相应的脚本程序。

表 1-5 命令行工具需要的用于设置环境变量的脚本

工具集	脚本
Visual C++	*vcvars32.bat*
Intel (Windows)	*iclvars.bat*
Intel (Linux)	*iccvars.sh* 或 *iccvars.csh*
Metrowerks (Mac OS X)	*mwvars.sh* 或 *mwvars.csh*
Metrowerks (Windows)	*cwenv.bat*
Comeau	与后端的工具集相同

编译和链接 *hello.cpp* 的命令如表 1-6 所示。要使这些命令正确运行，应使当前目录是含有 *hello.cpp* 的目录，且包含命令行编译器的目录应出现在 PATH 环境变量中。如果已在第1步中运行了脚本程序，那么后面的条件将自动满足。也有可能在你安装工具集时，安装程序就把包含命令行工具的目录添加到 PATH 中了。如果不是这样，你就得如表 1-7 所示的那样把目录添加到 PATH 中，或在命令行中指定完整的路径名。

表 1-6 只用一步就可完成 hello.cpp 编译和链接的命令

工具集	命令行
GCC	*g++ -o hello hello.cpp*
Visual C++	*cl -nologo -EHsc -GR -Zc:forScope -Zc:wchar_t -Fehello hello.cpp*
Intel (Windows)	*icl -nologo -EHsc -GR -Zc:forScope -Zc:wchar_t -Fehello hello.cpp*
Intel (Linux)	*icpc -o hello hello.cpp*
Metrowerks	*mwcc -wchar_t on -cwd include -o hello hello.cpp*
Comeau	*como -o hello hello.cpp*
Borland	*bcc32 -q -ehello hello.cpp*
Digital Mars	*dmc -Ae -Ar -I<dmcroot>/stlport/stlport -o hello hello.cpp*

www.TopSage.com

大家网 TopSage.com

表 1-7 把目录添加到 PATH 环境变量

外壳程序	命令行
bash, sh, ksh (Unix)	*export PATH=<directory>:$PATH*
csh, tsch (Unix)	*setenv PATH <directory>:$PATH*
cmd.exe (Windows)	*set PATH=<directory>;%PATH%*

例如，如果你使用的是 Microsoft Visual Studio .NET 2003，且是安装在 C 盘上的标准目录下，那么就可以切换到含有 *hello.cpp* 的目录下，输入如下命令：

```
> "C:\Program Files\Microsoft Visual Studio .NET2003\Vc7\bin\
vcvars32.bat"
Setting environment for using Microsoft Visual Studio .NET 2003 tools.
(If you have another version of Visual Studio or Visual C++ installed
and wish to use its tools from the command line, run vcvars32.bat for that
version.)
> cl -nologo -EHsc -GR -Zc:forScope -Zc:wchar_t -Fehello
hello.cpp
hello
```

于是，你现在就可以运行程序了：

```
> hello
Hello, World!
```

同样，如果你使用的是 Intel 9.0 for Linux，且如果是安装在标准位置 */opt/intel/cc/9.0* 下面，那么就可以打开 *bash* 外壳程序，切换到含有 *hello.cpp* 的目录，并输入如下命令：

```
$ . /opt/intel/cc/9.0/bin/iccvars.sh
$  icpc -o hello hello.cpp
$ ./hello
Hello, World!
```

讨论

环境变量就是由你的系统维护的字符串对，且可被正在运行的应用程序访问。命令行工具经常要提到环境变量以了解你的系统的详细信息，从而获得配置信息。如果没有这些信息，则必须从命令行中输入。遇到最多的环境变量是 PATH，它保存了一个目录列表，当在命令行中只输入可运行程序的名称（而不是完整路径名）时，操作系统会查找这个目录列表。在 Windows 系统下，当加载动态库时，也会查找 PATH 中的目录。

在 Unix 和 Windows 系统下，命令行工具都会使用环境变量，但在 Unix 系统下，往往会有一个占主导地位的 C++ 编译器，它所需要的环境变量已默认地设置为正确值了。但是，在 Windows 系统下，往往有多个 C++ 编译器。例如，两个不同的编译器肯定得查找不同的路径以获得它们的标准库头文件和已编译的运行时支持库。因此，对于 Windows 的工具集，通常要提供一些设置环境变量的脚本代码，以记录头文件和库的路径以及其他信息。

使用这种脚本程序的一种方法是，在运行任何命令行工具（如Visual C++和Intel 9.0 for Linux）之前，先运行脚本程序。也可以先设置好环境变量的参数，这样你就不用每次在运行命令行之前都必须运行脚本程序了。但是如何设置环境变量则取决于你的操作系统和你的外壳程序。然而，永久地修改你的环境变量并不是一个好主意，因为会有多个工具集可能含有名称相同的工具，从而使得在构建程序的过程中导致错误工具的调用。例如，如果你安装了多个版本的Visual C++，在使用命令行工具之前，必须确保运行正确的*vcvars32.bat*版本。另一个例子是，Visual C++和Digital Mars工具集都含有名为*link.exe*和*lib.exe*的工具。

现在，让我们来看看表1-7的命令行。记住，你只需关心与你所用的工具集相对应的行。通常，传递给编译器的信息可以归纳为四类：

- 输入文件名
- 输出文件名
- 要查找的文件的位置
- 通常的配置信息

在表1-6中，只有一个输入文件*hello.cpp*，且只是在命令行中输入文件名来传递给编译器。你可以把输入文件名随便放在任意位置，但不能出现在另一个命令行选项中间。在表1-7中，我把*hello.cpp*放在命令行的末尾了。

也只有一个输出文件*hello.exe*或*hello*（因操作系统而异）。但是，在这种情况下，把该文件传递给编译器的方式则取决于工具集。大多数工具集使用*-o <file>*来指定输出的可运行文件，但Visual C++和Intel for Windows使用*-Fe<file>*，Borland使用*-e<file>*.。注意，没有必要指定可执行文件的扩展名。

在表1-7中，属于第三类查找文件的位置是出现在Digital Mars命令行的唯一信息。由于STLPort库不是Digital Mars的内置标准库，必须用*-I*选项来告诉编译器到哪里去查找STLPort库文件。STLPort头文件位于Digital Mars安装路径的*/stlport/stlport*子目录下。在表1-7中我用*<dmcroot>/stlport/stlport*来指定该目录。关于*-I*选项的更多信息，请参见1.5节。

表1-7中的大多数命令行选项属于第四类：通常的配置信息。这些选项不是用于特定的文件，而是启用或禁用某些编译器特征。

- 选项*-nologo*（Visual C++和Intel for Windows）和*-q*（Borland）告诉编译器不显示其名字和版本号。这使得编译器的输出更易读。
- 选项*-EHsc*（Visual C++和Intel for Windows）和*-Ae*（Digital Mars）告诉编译器启用C++异常处理。

www.TopSage.com

- 选项 *-GR*（Visual C++ 和 Intel for Windows）和 *-Ar*（Digital Mars）告诉编译器启用运行时类型信息（RTTI）。
- 选项 *-Zc:wchar_t*（Visual C++ 和 Intel for Windows）和 *-wchar_t on*（Metrowerks）告诉编译器把 `wchar_t` 识别为内置类型。
- 选项 *-Zc:forScope*（Visual C++ 和 Intel for Windows）告诉编译器严格遵守 for-scoping 规则。
- 选项 *-cwd* include（Metrowerks）告诉编译器，在含有 `include` 指令的源文件目录中开始查找一个包含的头文件。

接下来，让我们来看看原始问题的另一种解决方案。这种解决方案不是用单个命令来编译和链接，而是把第 2 步分割成以下两部分：

2a. 输入一个命令，告诉编译器把程序编译成目标文件，但不进行链接。

2b. 输入一个命令，告诉链接器把在上一步中创建的目标文件生成可运行程序。

在前面的简单示例中，没有必要分别编译和链接。但是，分别编译和链接经常是需要的。因此，了解如何做是很重要的。例如，要创建一个静态库，就必须只编译而不链接，然后把结果目标文件传递给归档器。

分两步编译和链接的命令见表 1-8 和表 1-9 所示。在一些地方，目标文件扩展名 *o[bj]* 表明，单个命令行对 Windows 和 Unix 是合法的，但目标文件的扩展名除外。

表 1-8 用于编译但不链接 hello.cpp 的命令

工具集	命令行
GCC	*g++ -c -o hello.o hello.cpp*
Visual C++	*cl -c -nologo -EHsc -GR -Zc:forScope -Zc:wchar_t -Fohello hello.cpp*
Intel (Windows)	*icl -c -nologo -EHsc -GR -Zc:forScope -Zc:wchar_t -Fohello hello.cpp*
Intel (Linux)	*icpc -c -o hello.o hello.cpp*
Metrowerks	*mwcc -c -wchar_t on -cwd include -o hello.o[bj]hello.cpp*
Comeau	*como -c -o hello.o[bj] hello.cpp*
Borland	*bcc32 -c -q -o hello.obj hello.cpp*
Digital Mars	*dmc -c -Ae -Ar -I<dmcroot>/stlport/stlport -o hello.obj hello.cpp*

表 1-9 用于链接 hello.exe 或 hello 的命令

工具集	命令行
GCC	*g++ -o hello hello.o*
Visual C++	*link -nologo -out:hello.exe hello.obj*

www.TopSage.com

表1-9 用于链接hello.exe或hello的命令（续）

工具集	命令行
Intel (Windows)	*xilink -nologo -out:hello.exe hello.obj*
Intel (Linux)	*icpc -o hello hello.o*
Metrowerks	*mwld -o hello hello.o[bj]*
Comeau	*como --no_prelink_verbose -o hello hello.o[bj]*
Borland	*bcc32 -q -ehello hello.cpp*
Digital Mars	*link -noi hello.obj, hello.exe,NUL,user32.lib, kernel32.lib*

例如，要用GCC工具集来构建可执行程序hello，先切换到含有*hello.cpp*文件的目录下，然后输入命令：

```
$ g++ -c -o hello.o hello.cpp
$ g++ -o hello hello.o
```

现在，你就可以如下运行程序了：

```
$ ./hello
  Hello, World!
```

表1-9与表1-6几乎相同，只有两处不同。首先，选项*-c*用于告诉编译器只编译而不链接。其次，指定输出文件为目标文件*hello.obj*或*hello.o*，而不是可执行程序。大多数编译器使用*-o <file>*来指定输出文件，但Visual C++和Intel for Windows使用选项*-Fo<file>*。而且，除Visual C++和Intel for Windows外，所有编译器都要求指定目标文件的扩展名。

到现在为止，表1-9中的大多数命令行都应该很容易理解，因此这里只给出两个说明：

- Digital Mars链接器的命令行语法与众不同，它由6个逗号分隔的字段组成，用于指定输入文件的不同类型。现在，你只需知道第一个字段用于目标文件，第二个用于输出文件就可以了。选项-noi告诉链接器执行区分大小写的链接，这对C++程序是需要的。
- Borland链接器ilink32.exe使用与Digital Mars类似的语法。为了简化命令行，我使用的是bcc32.exe编译器来执行链接步骤。bcc32.exe会在后端调用ilink32.exe。

参见1.7节和1.15节。

1.3 从命令行创建静态库

问题

如何用命令行工具把C++源文件（如示例1-1所示的文件）创建成静态库？

www.TopSage.com

解决方案

首先，用编译器把源文件编译成目标文件。如果源文件包含了位于其他目录中的头文件，则需要用*-I*选项来告诉编译器到哪里去查找这些头文件，详细信息请参见1.5节。其次，用归档器把目标文件组合成静态库。

要编译示例1-1中的三个源文件，可以用表1-8中的命令行，并按需要修改输入和输出文件名。要把编译好的目标文件组合成静态库，可以使用表1-10中的命令。

表1-10 用于创建libjohnpaul.lib或libjohnpaul.a的命令

工具集	命令行
GCC (Unix)	ar ru libjohnpaul.a john.o paul.o johnpaul.o
Intel (Linux)	ranlib libjohnpaul.a
Comeau (Unix)	
GCC (Windows)	ar ru libjohnpaul.a john.o paul.o johnpaul.o
Visual C++	lib -nologo -out:libjohnpaul.lib john.obj paul.obj johnpaul.obj
Comeau (with Visual C++)	
Intel (Windows)	xilib -nologo /out:libjohnpaul.lib john.obj paul.obj johnpaul.obj
Metrowerks (Windows)	mwld -library -o libjohnpaul.lib john.obj paul.obj johnpaul.obj
Metrowerks (Mac OS X)	mwld -library -o libjohnpaul.a john.o paul.o johnpaul.o
Borland	tlib libjohnpaul.lib /u /a /C +john +paul +johnpaul
Digital Mars	lib -c -n libjohnpaul.lib john.obj paul.obj johnpaul.obj

例如，要用GCC把*john.cpp*、*paul.cpp*和*johnpaul.cpp*编译成目标文件，切换到**johnpaul**目录，输入如下命令来产生*john.o*、*paul.o*和*johnpaul.o*目标文件：

```
$ g++ -c -o john.o john.cpp
$ g++ -c -o paul.o paul.cpp
$ g++ -c -o johnpaul.o johnpaul.cpp
```

现在就可以如下把这些目标文件链接成一个静态库了：

```
$ ar ru libjohnpaul.a john.o paul.o johnpaul.o
$ ranlib libjohnpaul.a
```

讨论

在Unix系统中，使用GCC时，要用两个单独的命令来生成一个静态库。首先，运行归档器*ar*，然后运行名为*ranlib*的工具。*Ru*选项告诉*ar*，如果没有相同名字的归档文件，就把给定的目标文件添加到指定的归档器中，但只有当给定的目标文件比已有的文件更加新时，才会更新已有的归档文件。以前，在创建或更新归档后，就用*ranlib*工具来创建更新归档符号表（就是出现在不同目标文件中的符号索引）。现在，在很多系统中，归

www.TopSage.com

大家网 TopSage.com

档器自己负责创建或更新符号表，因此没有必要再运行*ranlib*了。GNU版本的*ar*就是如此。但是，在一些系统中，GCC编译器可能会与非GNU版本的*ar*一起使用，鉴于此原因，最好是运行*ranlib*以防万一。

从表1-10可以看出，Borland归档器*tlib*使用的语法稍有不同，目标文件前面的加号用于把这些目标文件添加到库中。对其他命令行你应该很容易理解。

注意： 对于其他工具集，通过传递恰当的命令行选项，链接器可以用作归档器。而对其他的工具集，则必须使用单独的归档器。

参见1.8节、1.11节和1.16节。

1.4 从命令行创建动态库

问题

如何使用命令行工具从示例1-2所示的C++源文件创建动态库？

解决方案

用以下步骤：

1. 用编译器把源文件编译成目标文件。如果使用的是Windows系统，用选项*-D*定义宏，确保输出动态库的符号。例如，要在示例1-2中创建动态库，需要定义宏GEORGERINGO_DLL。如果创建的是第三方动态库，安装指导说明会告诉你要定义什么宏。
2. 使用链接器从第1步中生成的目标文件创建动态库。

注意： 如果你的动态库依赖于其他的库，应告诉编译器到哪里去查找这些动态库，并把其他库的名字告诉链接器。1.5节将详细讨论。

执行第1步的基本命令如表1-8所示，你需要相应地修改输入和输出文件的名字。执行第2步的命令见表1-11所示。如果你所使用的工具集带有静态和动态运行库，则要告诉编译器和链接器使用动态链接运行库，具体见1.23节的介绍。

表1-11 创建动态库libgeorgeringo.so、libgeorgeringo.dll或libgeorgeringo.dylib的命令

工具集	命令行
GCC	*g++ -shared -fPIC -o libgeorgeringo.so george.o ringo.o georgeringo.o*

www.TopSage.com

表1-11 创建动态库libgeorgeringo.so、libgeorgeringo.dll或libgeorgeringo.dylib的命令(续)

工具集	命令行
GCC (Mac OS X)	*g++ -dynamiclib -fPIC -o libgeorgeringo.dylib george.o ringo.o georgeringo.o*
GCC (Cygwin)	*g++ -shared -o libgeorgeringo.dll -Wl,--out-implib,libgeorgeringo.dll.a -W1,--export-all-symbols -Wl,--enable-auto-image-base george.o ringo.o georgeringo.o*
GCC (MinGW)	*g++ -shared -o libgeorgeringo.dll -Wl,--out-implib,libgeorgeringo.a -W1,--export-all-symbols -Wl,--enable-auto-image-base george.o ringo.o georgeringo.o*
Visual C++	*link -nologo -dll -out:libgeorgeringo.dll -implib: libgeorgeringo.lib george.obj ringo.obj georgeringo.obj*
Intel (Windows)	*xilink -nologo -dll -out:libgeorgeringo.dll -implib: libgeorgeringo.lib george.obj ringo.obj georgeringo.obj*
Intel (Linux)	*g++ -shared -fPIC -lrt -o libgeorgeringo.so george.o ringo.o georgeringo.o georgeringo.obj*
Metrowerks (Windows)	*mwld -shared -export dllexport -runtime dm -o libgeorgeringo.dll -implib libgeorgeringo.lib george.obj ringo.obj georgeringo.obj*
Metrowerks (Mac OS X)	*mwld -shared -export pragma -o libgeorgeringo.dylib george.o ringo.o georgeringo.o*
CodeWarrior 10.0 (Mac OS X)	Consult the Metrowerks documentation
Borland	*bcc32 -q -WD -WR -elibgeorgeringo.dll george.obj ringo.obj georgeringo.obj* *implib -c libgeorgeringo.lib libgeorgeringo.dll*
Digital Mars	*dmc -WD -L/implib:libgeorgeringo.lib -o libgeorgeringo.dll george.obj ringo.obj georgeringo.obj user32.lib kernel32.lib*

注意： 到2005年9月为止，Comeau工具集不支持在Unix或Windows系统上创建动态库。但是，Comeau Computing公司正在开发对动态库的支持，预计到2005年年底，可以在一些Unix平台上（包括Linux）实现。

例如，要用Borland编译器把示例1-2的源文件编译成目标文件，假设Borland工具集的目录已包含在PATH环境变量中，切换到目录*georgeringo*，并输入以下命令：

```
> bcc32 -c -q -WR -o george.obj george.cpp
george.cpp:
> bcc32 -c -q -WR -o ringo.obj ringo.cpp
ringo.cpp:
> bcc32 -c -q -WR -DGERORGERINGO_DLL -o georgeringo.obj georgeringo.cpp
georgeringo.cpp:
```

www.TopSage.com

这里，编译器选项 *-WR* 用于指定动态运行库。这三个命令将输出目标文件 *george.obj*、*ringo.obj* 和 *georgeringo.obj*。接下来，输入命令：

```
> bcc32 -q -WD -WR -elibgeorgeringo.dll george.obj ringo.obj
georgeringo.obj
```

这将生成动态库 *libgeorgeringo.dll*。最后，输入命令：

```
> implib -c libgeorgeringo.lib libgeorgeringo.dll
```

将生成输入库 *libgeorgeringo.lib*。

讨论

如何处理动态库完全取决于操作系统和工具集。从程序开发人员的角度来说，两个最重要的区别如下。

符号可见性

动态库可以包含类、函数和数据的定义。在某些平台上，所有这些符号都能被使用动态库的代码访问，其他系统则为程序开发人员提供了访问这些符号的很好控制。能够确定哪些符号可见通常是有好处的，它使得程序开发人员更明确地控制库的公用接口，而且经常能提供很好的性能。但是，这也使得动态库的创建和使用更复杂。

对于大多数的Windows工具集，为了让使用动态库的代码能访问定义在动态库中的符号，则必须在创建动态库时显式地使其为可输出的（exported），使用动态库的可执行程序或动态库在创建时必须为可输入的（imported）。一些Unix工具集也提供了这种灵活性。对于一些平台的最新GCC版本、Mac OS X的Metrowerks和Linux的Intel工具集都是如此。但是，在一些情况下，则别无选择，只能使所有符号可见。

把库传递给链接器

在Unix系统中，当链接使用了动态库的代码时，动态库可以作为链接器的输入。在Windows系统中，除非使用GCC，否则动态库不能直接指定为链接器的输入，而是使用输入库或模块定义文件。

输入库和模块定义文件

输入库，粗略地说，就是含有运行时调用DLL的函数的信息的静态库。没有必要知道它们的工作原理，只要知道如何创建和使用它们就可以了。当你创建DLL时，大多数链接器自动地生成输入库，但在一些情况下，可能需要使用一个单独的工具，称为 *import librarian*。在表1-11中，我使用了Borland的 *implib.exe*，以避免使用Borland链接器 *ilink32.exe* 所要求的特殊命令行语言。

模块定义文件或 *.def* 文件是一个文本文件，它描述了由DLL输出的函数和数据。*.def* 文件可以手工编写或由工具自动生成。示例1-5为库 *libgeorgeringo.dll* 的一个示范 *.def* 文件。

www.TopSage.com

示例 1-5 libgeorgeringo.dll 的模块定义文件

```
LIBRARY             LIBGEORGERINGO.DLL

EXPORTS
    Georgeringo     @1
```

从 DLL 输出符号

从 Windows 的 DLL 中输出符号有两种标准的方法：

- 使用 DLL 文件的 `__declspec(dllexport)` 属性，当链接使用 DLL 的代码时，创建一个输入库。

 `__declspec(dllexport)` 属性应插在输出函数或数据声明的开头，如示例 1-6 所示。注意，`__declspec(dllexport)` 不是 C++ 语言的一部分，它是被大多数 Windows 编译器所实现的语言扩展。

- 创建一个用于描述被动态库输出的函数或数据的 *.def* 文件。

示例 1-6 __declspec(dllexport)属性

```
_ _declpec(dllexport) int m = 3;      // Exported data definition
extern _ _declpec(dllexport) int n;   // Exported data declaration
_ _declpec(dllexport) void f(  );     // Exported function declaration
class _ _declpec(dllexport) c {       // Exported class definition
   /* ... */
};
```

使用 *.def* 文件有一定的好处。例如，它可以使得 DLL 中的函数通过数字而不是名字来访问，从而可以减小 DLL 的大小。它还可以减小对预处理器指令（如示例 1-2 中的 *georgeringo.hpp* 头文件）的需要。但是，它有一些严重的缺点。例如，*.def* 文件不能用于输出类。而且，当你添加、删除或修改了 DLL 中的函数时，很难记得去更新 *.def* 文件。因此，我推荐你使用 `__declspec(dllexport)`。要学习 *.def* 文件的完整语法及用法，可以查询相关的工具集文档。

从 DLL 中输入符号

正如有两种方法从 DLL 中输出符号一样，输入符号也有两种方法：

- 在使用了 DLL 的源代码所包含的头文件中，使用 `__declspec(dllimport)` 属性，当链接使用了 DLL 的代码时，将把库输入到链接器。
- 当链接使用了 DLL 的代码时，指定一个 *.def* 文件。

与输入符号一样，我推荐你在源代码中使用 `__declspec(dllimport)` 属性，而不是 .def 文件。属性 `__declspec(dllimport)` 的使用就像前面介绍的 `__declspec(dllexport)` 属性一样。而且，与 `__declspec(dllexport)` 一样，`__declspec(dllimport)` 也不是 C++ 语言的一部分，而是由大多数 Windows 编译器的实现扩展。

www.TopSage.com

如果你选择使用__declspec(dllexport)和__declspec(dllimport)，就必须确保在创建DLL时使用了__declspec(dllexport)，在编译使用了DLL的代码时使用了__declspec(dllimport)。一种方法是使用两种头文件：一种用于创建DLL，另一种用于编译使用了DLL的代码。但是，这并不令人满意，因为要维护相同头文件的两个不同版本比较困难。

常用的方法是定义一个宏，当创建DLL时扩展为__declspec(dllexport)，否则就为__declspec(dllimport)。示例1-2使用宏GEORGERINGO_DECL就是为了这个目的。在Windows系统下，如果定义了宏GEORGERING_SOURCE，则把GEORGERINGO_DECL扩展为__declspec(dllexport)，否则扩展为__declspec(dllimport)。创建DLL *libgeorgeringo.dll*时定义GEORGERING_SOURCE，编译使用*libgeorgeringo.dll*的代码时则不用定义该宏，就可以获得想要的结果。

用GCC创建DLL

1.1节所介绍的用GCC工具集的Cygwin和MinGW创建DLL时与其他的Windows工具集有何不同。当用GCC创建DLL时，所有函数、类和数据都默认地为输出的。通过把*--no-export-all-symbols*选项传递给链接器，在源文件中使用属性__declspec-(dllexport)，或使用一个*.def*文件，就可以修改这种行为。在这三种情况下，除非使用*--export-all-symbols*强制让链接器输出所有符号，否则只有函数、类和数据被标识为__decl-spec(dllexport)或列举在*.def*文件中。

因此用GCC工具集创建DLL有两种方法：一种是像普通的Windows工具集一样，显式地使用__declspec输出符号；另一种是像Unix工具集一样，自动输出所有符号。在示例1-2和表1-11中使用的是后一种方法。如果你选用这种方法，为安全起见，应考虑使用选项*--export-all-symbols*，以防止你使用了含有__declspec(dllexport)的头文件。

GCC与其他Windows工具集不同的另一种方式是，不能把与DLL相关的输入库传递给链接器，而是传递给DLL本身。这通常比使用输入库更快。但是，这也可能产生问题，因为你的系统中可能存在某个DLL的多个版本，而你必须确保链接器选用的是正确的版本。在表1-11中，为了说明如何使用DLL输出输入库，我没有选用该特性。

注意：对于Cygwin，动态库DLL *xxx.dll*的输出库通常命名为*xxx.dll.a*，而对于MinGW则通常为*xxx.a*。这只是一种约定而已。

GCC 4.0的fvisibility选项

包括Linux和Mac OS X在内的多个平台上的GCC最新版本为程序开发人员提供了动态库输出的符号的很好控制：命令行选项*-fvisibility*可用来设置动态库的符号的默认可见性。*-fvisibility*选项有多个可选值，但最令人感兴趣的两个是*default*和*hidden*。粗略地

www.TopSage.com

大家网 TopSage.com

说，*default*可见性说明符号可以被其他模块中的代码访问，而*hidden*可见性则表示不能。要使符号输出为可选择的，在命令行中指定*-fvisibility=hidden*，然后用*visibility*属性来指定特定符号为可见的，如示例 1-7 所示。

示例 1-7 使用命令行选项 visibility 属性 -fvisibility=hidden

```
extern __attribute__((visibility("default"))) int m;      // exported
extern int n;                                              // not exported

__attribute__((visibility("default"))) void f();          // exported
void g();                                                  // not exported

struct __attribute__((visibility("default"))) S {};       // exported
struct T {};                                               // not exported
```

在示例 1-7 中，属性__attribute__((visibility("default")))与Windows代码中的__declspec(dllexport)作用相同。

使用visibility属性与使用__declspec(dllexport)和__declspec(dllimport)有相同的问题，当创建共享库时，你可能希望该属性为可见的，但当编译使用了共享库的代码时则不希望。与__declspec(dllexport)和__declspec(dllimport)一样，这个问题可以用预处理器来解决。例如，你可以如下修改示例 1-2 的头文件*georgeringo.hpp*，以利用 visibility 属性：

```
georgeringo/georgeringo.hpp 头文件
#ifndef GEORGERINGO_HPP_INCLUDED
#define GEORGERINGO_HPP_INCLUDED

// define GEORGERINGO_DLL when building libgerogreringo
# if defined(_WIN32) && !defined(__GNUC__)
#  ifdef GEORGERINGO_DLL
#   define GEORGERINGO_DECL __declspec(dllexport)
#  else
#   define GEORGERINGO_DECL __declspec(dllimport)
#  endif
# else // Unix
#  if defined(GEORGERINGO_DLL) && defined(HAS_GCC_VISIBILITY)
#   define GEORGERINGO_DECL __attribute__((visibility("default")))
#  else
#   define GEORGERINGO_DECL
#  endif
# endif

// Prints "George, and Ringo\n"
GEORGERINGO_DECL void georgeringo();

#endif // GEORGERINGO_HPP_INCLUDED
```

要使之有效，在支持*-fvisibility*选项的系统中创建时，必须定义宏HAS_GCC_VISIBILITY。

注意： Intel compiler for Linux 最新版本也支持 *-fvisibility* 选项。

Metrowerks for Mac OS X 的符号可见性

Metrowerks for Mac OS X 提供了几个选项，用于从动态库中输出符号。当使用

www.TopSage.com

CodeWarrior IDE时，可以使用符号输出文件，该文件与Windows系统下的*.def*作用相同。你也可以选择使用选项*-export all*输出所有符号，当从命令行创建时这是默认的。我推荐的方法是在源代码中使用`#pragma export`来标识要输出的函数，当在命令行中创建动态库时指定*-export pragma*。`#export pragma`的使用见示例1-2所示：在头文件中，在要输出的函数组的前面使用`#pragma export`，在其后面再使用`#export pragma off`。如果你想让你的代码在Metrowerks之外的工具集上也可以使用，就必须在`#ifdef/#endif`指令之间使用`#pragma export`，如示例1-2所示。

命令行选项

让我们来快速浏览一下表1-11中使用的选项。每个命令行指定：

- 输入文件名：*george.obj*、*ringo.obj*和*georgeringo.obj*。
- 要创建的动态库的名称。
- 在Windows系统中，输入库的名称。

另外，链接器要求有一个选项来告诉它是创建动态库而不是可运行程序。大多数链接器使用选项*-shared*，但Visual C++和Intel for Windows使用的是*-dll*，Borland和Digital Mars使用的是*-WD*，GGC for Mac OS X使用的是*-dynamiclib*。

表1-11中有几个选项可以使得在运行时使用动态库更有效。例如，一些Unix链接器应使用选项*-fPIC*（GCC和Intel for Linux）来告诉它生成与位置无关的代码。该选项更像是在某些系统中，多个进程共享动态库代码的单个副本。如果没有指定这个选项，则会导致链接器错误。同样，在Windows系统中，GCC链接器选项*--enable-auto-image-base*使得它不像是操作系统在同一个地方要加载两个动态库，使用该选项可以加快DLL的加载速度。

注意：可以通过编译器，使用编译器选项*-Wl,<option>* to *g++*（W后面的是小写字母l）把命令行选项传递给GCC链接器。

大多数其他的选项用于指定运行时库变量，1.23节将详细介绍。

参见1.9节、1.12节、1.17节、1.19节和1.23节。

1.5 从命令行创建复杂的应用程序

问题

如何使用命令行工具创建一个使用多个静态库和动态库的可执行程序？

www.TopSage.com

解决方案

首先创建应用程序所需的静态库和动态库。如果是第三方的库，遵循随库发布的指导，否则，按 1.3 节和 1.4 节所介绍的方法创建。

接下来，像从命令行创建简单的“Hello, World”程序那样，把应用程序的 *.pp* 文件编译成目标文件。你可能需要使用 *-I* 选项，告诉编译器到哪里去查找应用程序所需的头文件，如表 1-12 所示。

表 1-12 指定查找头文件的目录

工具集	选项
所有工具集	*-I<directory>*

最后，使用链接器从目标文件和库输出可执行程序。对每一个库，你必须要么提供完整的路径名，要么告诉编译器到哪里去查找它。

在上面的每一步中，如果你所使用的工具集带有运行库的静态和动态变量，且程序至少使用了其中一个动态库，就必须告诉编译器和链接器动态地使用已链接的运行库，如 1.23 节所示。

表 1-13 显示了链接示例 1-3 的 hellobeatles 程序的命令。它假设：

- 当前目录是 *hellobeatles*。
- 静态库 *libjohnpaul.lib* 和 *libjohnpaul.a* 创建在 *johnpaul* 目录中。
- 动态库 *georgeringo.dll*、*georgeringo.so* 和 *georgeringo.dylib* 及其输入库（如果有）创建在 *georgeringo* 目录中。

注意： 由于 Comeau 不能创建动态库，见 1.4 节，在表 1-13 的 Comeau 项中，假设 *libgeorgeringo* 是作为一个静态库（而不是动态库）来创建的。要把 *libgeorgeringo* 创建为静态库，需要从示例 1-2 的 `georgeringo()` 函数中删除修饰符 `GEORGERINGO_DECL`。

表 1-13 链接应用程序 hellobeatle.exe 的命令

工具集	输入文件	命令行
GCC (Unix)	*hellobeatles.o* *libjohnpaul.a* *libgeorgeringo.so*	*g++ -o hellobeatles hellobeatles.o -L../johnpaul -L../georgeringo -ljohnpaul lgeorgeringo* 或 *g++ -o hellobeatles hellobeatles.o ./johnpaul/libjohnpaul.a ../georgeringo/libgeorgeringo.so*

www.TopSage.com

表 1-13 链接应用程序 hellobeatle.exe 的命令（续）

工具集	输入文件	命令行
Intel (Linux)		*icpc -o hellobeatles hellobeatles.o -L../johnpaul -L../georgeringo -ljohnpaul georgeringo* 或 *icpc -o hellobeatles hellobeatles.o ../johnpaul/libjohnpaul.a ../georgeringo/libgeorgeringo.so*
Comeau (Unix)		*como --no_prelink_verbose -o hellobeatles hellobeatles.o -L../johnpaul -L../georgeringo -ljohnpaul lgeorgeringo* 或 *como --no_prelink_verbose -o hellobeatles hellobeatles.o ../johnpaul/libjohnpaul.a ../georgeringo/libgeorgeringo.a*
GCC (Mac OS X)	*hellobeatles.o* *libjohnpaul.a* *libgeorgeringo.dylib*	*g++ -o hellobeatles hellobeatles.o -L../johnpaul -L../georgeringo -ljohnpaul lgeorgeringo* 或 *g++ -o hellobeatles hellobeatles.o ../johnpaul/libjohnpaul.a ../georgeringo/libgeorgeringo.dylib*
Metrowerks (Mac OS X)		*mwld -o hellobeatles hellobeatles.o -search -L../johnpaul -search -L../georgeringo -ljohnpaul lgeorgeringo* 或 *mwld -o hellobeatles hellobeatles.o ../johnpaul/libjohnpaul.a ../georgeringo/libgeorgering.dylib*
GCC (Cygwin)	*hellobeatles.o* *libjohnpaul.a* *libgeorgeringo.dll.a*	*g++ -o hellobeatles hellobeatles.o -L../johnpaul -L../georgeringo -ljohnpaul lgeorgeringo* 或 *g++ -o hellobeatles hellobeatles.o ../johnpaul/libjohnpaul.a ../georgeringo/libgeorgeringo.dll.a*

www.TopSage.com

表 1-13 链接应用程序 hellobeatle.exe 的命令（续）

工具集	输入文件	命令行
GCC (MinGW)	*hellobeatles.o* *libjohnpaul.a* *libgeorgeringo.a*	*g++ -o hellobeatles hellobeatles.o -L../johnpaul -L../georgeringo -ljohnpaul lgeorgeringo* 或 *g++ --o hellobeatles hellobeatles.o ../johnpaul/libjohnpaul.a ../georgeringo/libgeorgeringo.a*
Visual C++	*hellobeatles.obj* *libjohnpaul.lib* *libgeorgeringo.lib*	*link -nologo -out:hellobeatles.exe -libpath:../johnpaul -libpath:../georgeringo libjohnpaul.lib libgeorgeringo.lib hellobeatles.obj*
Intel (Windows)		*xilink -nologo -out:hellobeatles-libpath:../johnpaul -libpath:../georgeringo libjohnpaul.lib libgeorgeringo.lib hellobeatles.obj*
Metrowerks (Windows)		*mwld-o hellobeatles -search -L../johnpaul libjohnpaul.lib -search -L../georgeringo libgeorgeringo.lib hellobeatles.obj*
Metrowerks (Mac OS X)		*mwld -o hellobeatles hellobeatles.o -search -L../johnpaul -search -L../georgeringo libjohnpaul.a libgeorgeringo.dylib*
CodeWarrior 10.0 (Mac OS X)		参见 Metrowerks 文档
Borland		*bcc32 -q -WR -WC -ehellobeatles -L.../johnpaul -L.../georgeringo libjohnpaul.lib libgeorgeringo.lib hellobeatles.obj*
Digital Mars		*link -noi hellobeatles.obj, hellobeatles.exe,NUL,user32.lib kernel32.lib ..\johnpaul\ ..\georgeringo\ libjohnpaul.lib libgeorgeringo.lib,,* 或 *link -noi hellobeatles.obj, hellobeatles.exe,NUL,user32.lib kernel32.lib ..\johnpaul z\libjohnpaul.lib..\georgeringo \libgeorgeringo.lib,,*

www.TopSage.com

大家网 TopSage.com

表 1-13 链接应用程序 hellobeatle.exe 的命令（续）

工具集	输入文件	命令行
Comeau (Windows)	*hellobeatles.obj* *libjohnpaul.lib* *libgeorgeringo.lib*	*como --no_prelink_verbose -o hellobeatles ../johnpaul/ libjohnpaul.lib ../georgeringo/ libgeorgeringo.lib hellobeatles.obj*

例如，如果你使用的是 Microsoft Visual Studio .NET 2003，且如果它是安装在 C 盘的标准位置中，就可以在命令行中切换到 *hellobeatles*，输入如下命令来创建 *hellobeatles.exe*：

```
> "C:Program Files\Microsoft Visual Studio .NET 2003\VC\bin\
vcvars32.bat"
Setting environment for using Microsoft Visual Studio 2005 tools.
(If you have another version of Visual Studio or Visual C++ installed
and wish to use its tools from the command line, run vsvars32.bat for
that version.)
> cl -c -nologo -EHsc -GR -Zc:forScope -Zc:wchar_t -MD -I..
-Fohellobeatles hellobeatles.cpp
hellobeatles.cpp
> link -nologo -out:hellobeatles.exe -libpath:../johnpaul
-libpath:../georgeringo libjohnpaul.lib libgeorgeringo.lib
hellobeatles.obj
```

讨论

查找被包含的头文件

-I 选项用于指定一个包含路径。当编译器（实际上是预处理器）碰到这种形式的包含指令时：

```
#include "file"
```

通常，它首先是试图通过解析与源文件相关的给定路径名来查找所引用的文件。如果不成功，则将试图按 *-I* 选项中指定的目录之一中查找文件，然后再去工具集目录中查找。工具集目录通常可以使用环境变量来配置。

这种情况类似于使用尖括号来指定被包含的头文件，如下所示：

```
#include <file>
```

但在这里，编译器往往不会解析与源文件相关的给定路径名。

把库传递给链接器

表 1-13 的命令行有几个令人感兴趣的方面。

在 Windows 系统中，链接器的输入由目标文件、静态库和输入库组成；在 Unix 系统中，则由目标文件、静态库和动态库组成。

www.TopSage.com

在 Windows 和 Unix 系统中，都有两种方法把库传递给链接器：

- 通过在命令行中指定路径名。
- 指定库的名称和查找该库的路径。

表 1-13 对这两种方法都进行了说明。

查找库的路径通常可以在命令行中指定。大多数链接器使用选项 *-L<directory>* 来指定，但 Visual C++ 和 Intel for Windows 使用的是 *-lipath: <directory>*，而 Metrowerks 使用的是 *-search -L<directory>*。Digital Mars 链接器允许在命令行中把库查找路径与库文件一起列举出来，并使用反斜杠把查找路径与库文件名区分开来，它还要求把反斜杠用作路径名的分隔符。

注意：在 Windows 系统中，Comeau 没有提供用于指定库查找路径的选项。

除可以显式地指定路径外，链接器通常还会从工具集目录列表中查找，其中工具集列表可以使用环境变量来配置。在 Windows 系统中，该目录列表往往包括工具集安装目录的 lib 子目录。因此，如果你把 *.lib* 文件复制到该目录中，就可以在命令行中使用名字，而无需指定查找的路径。如 1.25 节所示，如果你使用这种方法，就可以给链接器传递库的任何信息了。

在 Unix 和 Windows 系统中，为链接器指定库名的方法有所不同。在 Windows 系统中，需要指定库的完整名称，包括文件扩展名。在 Unix 系统中（在 Windows 中使用工具集 GCC 也是如此），是使用 *-l* 选项后跟库名来指定库的，文件扩展名和 *lib* 前缀都可以去掉。这意味着库名必须以 *lib* 开头，使链接器可以自动地查找到。更令人感兴趣的是，它允许链接器可以在库的多个版本中进行选择（除一些系统中之外，链接器可以基于文件名后面的 *.so* 在库的多个版本中进行选择）。

注意：Metrowerks 支持 Windows 和 Unix 风格的指定库名。

最后应注意的是，Unix 链接器对命令行中目标文件和静态库的指定顺序很敏感：如果某个静态库和目标文件引用了在另一个静态库和目标文件中定义的符号，则在命令行中第一个文件必须先于第二个文件出现。为了解决循环依赖的问题，有时候需要指定某个库和目标文件多次。另一种解决方案是把目标文件和静态库序列传递给链接器，并用-(和-)括起来。这将导致链接器反复查找文件，指定所有引用都被解析。应尽可能避免使用这种方法，因为它会明显地降低性能。

运行应用程序

如果应用程序使用了工具集运行库的动态变量，当运行应用程序时，该运行库必须是可

www.TopSage.com

用的，并且它必须是位于操作系统可以自动找到的位置。通常，这意味着动态运行库必须放在应用程序相同的目录中，或系统特定的目录列表中。在开发Windows程序时，这是比开发Unix程序更需要关注的问题，因为在Unix系统中，相应的运行库往往已经安装在正确的位置中。关于不同工具集的动态运行库的名称见1.23节的介绍。

参见1.10节、1.13节、1.18节和1.23节。

1.6 安装Boost.Build

问题

如何获得和安装Boost.Build?

解决方案

可以参考*www.boost.org/boost-build2*的Boost.Build文档或遵循以下这些步骤：

1. 访问Boost的主页*www.boost.org*，并找到Boost的SourceForge下载区中的下载链接。
2. 下载并解压缩*boost*包的最新版本或*boost-build*包的最新版本。前者包含了Boost库的全部集合，而后者是Boost.Build的单独发布版。将解压文件放在合适的永久位置。
3. 下载并解压缩适用于你的平台的*boost-jam*包的最新版本，该包含有预先创建好的*bjam*可运行程序。如果没有适用于你的平台的*boost-jam*包，按照在上一步中下载的包所提供的指示，从源文件创建可运行程序。
4. 把*bjam*复制到PATH环境变量中的相应位置。
5. 把环境变量BOOST_BUILD_PATH设置为Boost.Build根目录。如果你已在第1步中下载了*boost*包，那么该根目录是Boost安装目录的子目录*tools/build/v2*，否则就是目录*boost-build*。
6. 通过编辑Boost.Build根目录中的配置文件*user-config.jam*，可以为你的工具集和库配置Boost.Build。

讨论

使用Boost.Build最困难的就是下载和安装它。Boost也许最终会提供一个图形安装工具，但现在，你必须执行以上步骤。

上面第5步的目的是帮助创建工具*bjam*找到创建系统的根目录。但这一步不是严格要求的，因为还有另一种方法来完成这一工作：只需创建一个含有如下一行的文件*boost-build.jam*：

www.TopSage.com

```
boost-build boost-build-root ;
```

并把它放置在你的项目的根目录或该项目的任一父目录中。如果你希望把Boost.Build与你的源代码一起发布，第二种方法可能会更好些，因为这将使得终端用户的安装更容易。

第 6 步可能是最复杂的，但实际上它往往是很简单的。如果你只是安装了该工具集的一个版本，且是安装在标准位置，*user-config.jam* 文件包含下面形式的一行就足够了：

```
using <toolset> ;
```

例如，如果你使用的是 Visual C++，下面一行就够了：

```
using msvc ;
```

如果使用的是 GCC，下面这样写就可以了：

```
using gcc ;
```

如果你安装了工具集的多个版本，或者工具集不是安装在标准位置，情况就稍微复杂一些了。如果工具集不是安装在标准位置，你需要把调用工具集编译器的命令作为第三个参数传递给 `using`，告诉 Boost.Build 到哪里去找它。例如：

```
using msvc : "C:/Tools/Compilers/Visual Studio/Vc7/bin/cl" ;
```

如果你安装了工具集的多个版本，可以把工具集名作为第一个参数，版本号作为第二个参数，编译器命令作为第三个参数传递给 `using`。例如，你可以如下配置 Intel 工具集的两本版本：

```
using intel : 7.1 : "C:/Program Files/Intel/Compiler70/IA32/Bin/icl" ;
using intel : 8.0 : "C:/Program Files/Intel/CPP/Compiler80/IA32/Bin/icl" ;
```

本章所介绍的 7 种 Boost.Build 工具集如表 1-14 所示。

表 1-14 Boost.Build 工具集名

工具集	名称
GCC	`gcc`
Visual C++	`msvc`
Intel	`intel`
Metrowerks	`cw`
Comeau	`como`
Borland	`borland`
Digital Mars	`dmc`

www.TopSage.com

1.7 使用 Boost.Build 创建“Hello, World”应用程序

问题

如何使用 Boost.Build 创建如示例 1-4 所示的简单的 "Hello, World" 程序？

解决方案

在可运行程序所在的目录中创建一个名为 *Jamroot* 的文本文件，并创建所有相应的中间文件。在 *Jamroot* 文件中，如下调用两种规则。首先，调用 exe 规则以声明一个可运行程序，把 *.cpp* 文件指定为源文件。接着，调用 instal 规则，指定可运行程序名和安装目录的位置。最后，运行 *bjam* 来创建你的程序。

例如，要从示例 1-4 的文件 *hello.cpp* 创建可运行程序 *hello* 或 *helo.exe*，在含有 *hello.cpp* 文件的目录中，创建一个包含以下内容的 *Jamroot* 文件，如示例 1-8 所示。

示例 1-8 项目 hello 的 Jamfile 文件

```
# jamfile for project hello

exe hello : hello.cpp ;

install dist : hello : <location>. ;
```

接着，切换到含有 *hello.cpp* 和 *Jamroot* 的目录，输入一些命令：

```
> bjam hello
```

该命令在当前目录的子目录中创建可运行程序 *hello* 或 *hello.exe*。最后，输入命令：

```
> bjam dist
```

该命令把所创建的可运行程序复制到location属性指定的目录中，这里就是当前目录。

本书写作时，Boost.Build 开发人员正准备发布 Boost.Build version 2 正式版。当你阅读本书时，第2版可能已经发布了。如果还没有发布，你可以通过给 bjam 传送命令行选项 --v2 来启用本章所介绍的这种行为。例如，不是输入 bjam hello，而是输入 bjam --v2 hello。

讨论

文件 *Jamroot* 是 *Jamfile* 的一个样本。小型的C++源文件集可能用单个的Jamfile来管理，但大型的代码库则可能往往需要多个Jamfiles，按分层形式进行组织。每个 *Jamfile* 文件放置在各自的目录中，对应于各自的项目。大多数Jamfile文件就命名为 *Jamfile*，但最高级的 Jamfile（即位于项目根目录中的 Jamfile）命名为 *Jamroot*。由最高级 *Jamfile* 文件定义的项目称为项目根。除项目根之外的所有项目都有一个父项目。

分层设计功能很强大。例如，它很容易适合于项目及其子项目的要求（如线程支持）。

www.TopSage.com

大家网 TopSage.com

每个项目是目标文件的集合。该集合由调用规则（如exe规则、install规则）声明。大多数目标对应于二进制文件，或者更正确地说，是相关二进制文件的集合，如应用程序的调试版和发布版。

exe规则用于声明一个可运行程序目标。这种规则的一种使用形式如示例1-9所示。

示例 1-9 exe规则的使用

```
exe target-name
    : sources
    : requirements
    : default-build
    : usage-requirements
    ;
```

这里，*target-name*指定可运行程序名，*sources*指定源文件和库的列表，*requirements*指定将应用到目标的属性，*default-build*指定应用到目标的属性，*usage-requirements*指定属性，该属性可以应用到与该目标依赖的所有其他目标。

属性值以*<feature>value*的形式指定。例如，要声明一个利用线程支持来创建的可运行程序，可以如下编写：

```
exe hello
    : hello.cpp
    : <threading>multi
    ;
```

除非你为这些参数指定了值，否则，按照Boost.Build规则，不必用分号来隔开末尾的参数。

表1-15举了几种常用特性及其可能的值。

表 1-15 常用的 Boost.Build 特性

特性	值	作用
include	*Path*	指定一个包含路径
define	*name[=value]*	定义一个宏
threading	multi或single	启用或禁用线程支持
runtime-link	static或shared	指定运行库链接[注]
variant	debug或release	创建调试版或发布版

注：参见1.23节。

当创建可执行程序或相应的静态库或动态库时，与目标相对应的文件生成在含有Jamfile文件的目录的子目录中。该目录的相对路径名与工具集和创建配置有关，但总是以*bin*开头。例如，由示例1-8创建的可运行程序将在目录*bin/msvc/debug*中生成。

为简单起见，我推荐你把示例1-8的Jamfile生成在与源文件*hello.cpp*相同的目录中。但

www.TopSage.com

是，在实际的项目中，你可能希望把源文件和二进制文件保存在单独的目录中。在示例1-8中，你可以把Jamfile文件放在你喜欢的任意地方，你只要调整路径名，这样它就可以指向文件hello.cpp。

install规则命令Boost.Build把一个或多个文件（以文件名或主目标名指定）复制到指定的位置。示例1-10显示了这个规则的使用。

示例1-10 install规则的使用

```
install target-name
    : files
    : requirements
    : default-build
    : usage-requirements
    ;
```

这里，*target-name*是所声明的目标名，*files*是一个或多个要复制的文件和目标。其余的参数*requirements*、*default-build*和*usage-requirements*与示例1-9中的含意相同。

要复制的文件的位置可以以目标名和目标需求的location属性的值的形式来指定。例如，在示例1-8中，可以如下编写install目标：

```
install . : hello ;
```

然后就可以如下安装可运行程序：

```
> bjam .
```

但是，示例1-8更好，因为已命名的目标比文件路径名更易记。

最好，让我们来简单地看看*bjam*命令的语法。要用默认的工具集创建目标xxx，输入命令：

```
> bjam xxx
```

要用工具集*yyy*创建目标*xxx*，输入命令：

```
> bjam xxx toolset=yyy
```

要用工具集*yyy*的*vvv*版本创建目标*xxx*，输入命令：

```
> bjam xxx toolset=yyy-vvv
```

要从命令行创建标准库*zzz*，使用的语法为：

```
> bjam xxx stdlib=zzz
```

你也可以在命令行中输入多个目标名来同时创建多个目标，通过在给定项目中不指定目标来创建所有目标。因此，可以如下输入命令来从示例1-9创建和安装可运行程序：

```
> bjam
```

要在创建过程中删除所生成的文件（包括可执行程序），输入：

www.TopSage.com

```
> bjam --clean
```

注意： *<feature>value* 形式的属性可以在命令行中以 *feature=value* 的形式来指定。

参见 1.2 节和 1.15 节。

1.8 使用 Boost.Build 创建静态库

问题

如何使用 Boost.Build 从示例 1-1 所示的源文件创建静态库？

解决方案

在将要生成的静态库所在的目录中，创建一个 *Jamroot* 文件。在 *Jamroot* 文件中，调用 `lib` 规则来定义库目标，把 *.cpp* 文件指定为 sources 的值，属性 `<link>static` 为 requirement 的值。添加一个 usage-requirement，其值为 `<include>`*path* 形式，用来指定库的包含目录。你还需要添加一、两个值为 `<include>`*path* 形式的 requirements，告诉编译器到哪里去查找被包含的头文件。最后，从含有 *Jamroot* 文件的目录中运行 *bjam*，如 1.7 节所示。

例如，要从示例 1-1 的源代码创建静态库，其 *Jamroot* 文件可以如示例 1-11 所示：

示例 1-11 用于创建静态库 libjohnpaul.lib 或 libjohnpaul.a 的 Jamroot 文件

```
# Jamfile for project libjohnpaul

lib libjohnpaul
    : # sources
      john.cpp paul.cpp johnpaul.cpp
    : # requirements
      <link>static
    : # default-build
    : # usage-requirements
      <include>..
    ;
```

要创建该库，输入命令：

```
> bjam libjohnpaul
```

讨论

`lib` 规则用来声明一个表示静态库或动态库的目标。它采用与 `exe` 规则一样的形式，见示例 1-9 所示。`<link>static` 说明你的目标总是创建为一个静态库，如果你想创建静态库或动态库，可以省略掉 `<link>static`。是创建静态库还是动态库，可以在命令行指定，或在与库目标有关的需求中指定。例如，如果在示例 1-11 中省略了 `<link>static`，输入以下命令就可以把目标 `libjohnpaul` 创建为静态库：

```
> bjam libjohnpaul link=static
```

www.TopSage.com

但是，要编写既可以创建成静态库又可以创建成动态库的源文件要求很严格，具体请参见1.9节。

参见1.3节、1.11节和1.16节。

1.9 使用Boost.Build创建动态库

问题

如何使用Boost.Build从示例1-2所示的C++源文件创建动态库？

解决方案

在将要生成的动态库或输入库所在的目录中，创建一个*Jamroot*文件。在*Jamroot*文件中，调用lib规则来定义库目标，把*.cpp*文件指定为sources的值，属性<link>static为requirement的值。添加一个usage-requirement，其值为<include>*path*形式，用来指定库的包含目录。如果源文件还包含了来自其他库的头文件，你还需要添加一、两个值为<include>*path*形式的requirements，告诉编译器到哪里去查找被包含的头文件。最后，从含有*Jamroot*文件的目录中运行*bjam*，如1.7节所示。

例如，要从示例1-2的源文件创建动态库，在目录*georgeringo*中生成名为*Jamroot*的文件，如示例1-12所示。

示例1-12 用于创建动态库georgeringo.so、georgeringo.dll或georgeringo.dylib的Jamroot文件

```
# Jamfile for project georgringo

lib libgeorgeringo
    : # sources
      george.cpp ringo.cpp georgeringo.cpp
    : # requirements
      <link>shared
      <define>GEORGERINGO_DLL
    : # default-build
    : # usage-requirements
      <include>..
    ;
```

要创建该库，输入命令：

```
> bjam libgeorgeringo
```

讨论

正如1.8节所介绍的，lib规则用来声明一个表示静态库或动态库的目标。<link>shared说明你的目标总是创建为一个动态库。如果你想创建静态库或动态库，可以省略掉<link>static。是创建静态库还是动态库，可以在命令行指定，或在与库

www.TopSage.com

目标有关的需求中指定。请重新编写示例1-2，使得它既可以创建为静态库，又可以创建为动态库。

参见 1.4 节、1.12 节、1.17 节和 1.19 节。

1.10 使用 Boost.Build 创建复杂应用程序

问题

如何使用 Boost.Build 来创建一个使用多个静态库和动态库的可执行程序？

解决方案

可以遵循以下步骤：

1. 为可执行程序所依赖的每个库（除非该库是预先创建好的二进制文件）生成一个 Jamfile 文件，如 1.8 节和 1.9 节所示。
2. 在将要生成可执行程序的目录中创建一个 *Jamroot* 文件。
3. 在 *Jamroot* 文件中，调用 `exe` 规则声明可执行程序目标。指定 *.cpp* 文件和可执行程序依赖的库目标作为 sources 的值。同样，如果需要，添加 `<include>path` 形式的属性作为 sources 的值，用来告诉编译器到哪里去查找库的头文件。
4. 在 *Jamroot* 文件中，调用 `install` 规则，把属性 `<install-dependencies>on`、`<install-type>EXE` 和 `<install-type>SHARED_LIB` 指定为 requirements 的值。
5. 从含有 *Jamroot* 文件的目录中运行 *bjam*，如 1.7 节所示。

例如，要从示例 1-3 的源文件构建可执行程序，先在 *hellobeatles* 目录中生成一个名为 *Jamroot* 的文件，如示例 1-13 所示。

示例 1-13 用于创建可执行程序 hellobeatles.exe 或 hellobeatles 的 Jamfile 文件

```
# Jamfile for project hellobeatles

exe hellobeatles
    : # sources
      ../johnpaul//libjohnpaul
      ../georgeringo//libgeorgeringo
      hellobeatles.cpp
    ;

install dist
    : # sources
      hellobeatles
    : # requirements
      <install-dependencies>on
      <install-type>EXE
      <install-type>SHARED_LIB
      <location>.
    ;
```

www.TopSage.com

大家网 TopSage.com

现在，在*hellobeatles*目录中输入：

```
> bjam hellobeatles
```

这首先创建两个*hellobeatles*依赖的项目，然后创建目标hellobeatles。最后，输入：

```
> bjam dist
```

它将把可执行程序*hellobeatles*和动态库*georgeringo*复制到含有*hellobeatles.cpp*文件的目录中。

正如1.5节所讨论的那样，在运行*hellobeatles*之前，应把工具集的动态运行库放置在操作系统可以查找到的位置。

讨论

库目标

使用*path//target-name*表示法，可以把可执行程序依赖的库目标指定为sources的值。在1.8节和1.9节中介绍了如何利用Boost.Build为库声明目标，以从源文件创建库。但是，如果库是预先创建好的二进制文件，就可以如下声明目标：

```
lib target-name
    :
    : <file>file-name
    ;
```

正如1.7节所介绍的那样，最主要的目标所对应的不是单个文件，而是诸如可执行程序的调试版和发布版之类的相关文件集合。要为具有多个变种的预创建库声明目标，可以如下表示：

```
lib target-name
    :
    : <file>file-name requirements
    ;

lib target-name
    :
    : <file>other-file-name other-requirements
    ;
```

例如，可以如下声明预创建库的调试版和发布版：

```
lib cryptolib
    :
    : <file> ../libraries/cryptolib/cryptolib_debug.lib
      <variant>debug
    ;

lib cryptolib
    :
    : <file> ../libraries/cryptolib/cryptolib.lib
      <variant>release
    ;
```

www.TopSage.com

如果预创建库是位于链接器可以自动查找到的目录中，如1.5节所示，就可以如下声明目标：

```
lib target-name
    :
    : <name>library-name
    ;
```

这里，*library-name*是要传递给链接器的库名，它可能与实际的文件有所不同，见 1.5 节。要告诉链接器在某个目录中查找文件，可以如下编写：

```
lib target-name
    :
    : <name>library-name
      <search>library-path
    ;
```

安装

安装一个复杂的应用程序时，需要一些其他的可执行程序和它依赖的动态库。你可以使用与安装有关的特性，用来指定高级可执行程序目标和应安装的动态库，而不是单个地指定这些文件。在示例 1-3 中，需求属性 <install-dependencies>on 用于启用与安装有关的特性，<install-type>EXE和<install-type>SHARED_LIB则告诉Boost.Build安装所有依赖的可执行程序和共享库。其他与安装相关的特性值包括LIB和IMPORT_LIB。

项目组织结构

创建 *hellobeatles* 可执行程序的三个 *Jamfiles* 都是命名为 *Jamroot*。在简单的示例中，这种方法还行，但通常最好是以分层的形式来组织 *Jamfiles* 集合，其中以一个最高层的 *Jamfile* 来定义项目根。以这种方法来组织项目，可以充分利用 Boost.Build 的特性，例如，允许一些属性被子项目继承。这上面的示例中，要实现这些的方法是，在目录 *johnpaul*、*georgeringo* 和 *hellobeatles* 中，把 *Jamfiles* 文件名修改为 *Jamfile*，并在父目录的 *Jamroot* 文件中添加如下内容：

```
# jamfile for example application

build-project hellobeatles  ;
```

build-project 规则告诉 *bjam* 创建给定的项目，该项目可以通过路径名或符号标识符来指定。如果你切换到含有 *Jamroot* 文件的目录并运行 *bjam*，则会创建三个子项目。

参见 1.5 节、1.13 节和 1.18 节。

1.11 使用 IDE 创建静态库

问题

如何使用 IDE 把示例 1-1 的 C++ 源代码创建成静态库？

www.TopSage.com

解决方案

基本轮廓如下所示：

1. 生成一个新项目，指定你要创建静态库，而不是可执行程序或动态库。
2. 选择一个创建配置（例如，调试版或发布版，单线程或多线程）。
3. 指定库名以及它所在的目录。
4. 把源文件添加到该项目中。
5. 如果有必要，指定一个或多个目录，告诉编译器到那里去查找所包含的头文件。见1.13节所示。
6. 创建该项目。

该轮廓中的步骤可能会因不同的IDE而有所变化。例如，对于某些IDE，其中有几步合成了一步，步骤的顺序也有所不同。第2步在1.21节、1.22节和1.23节进行了详细介绍。这里，你尽可能使用默认设置。

例如，这里来看看如何使用Visual C++ IDE从示例1-1的源代码创建静态库。

从File菜单选择New → Project，在左边面板中选择Visual C++ Projects，选择Win32 Console Application，并输入*libjohnpaul*作为项目名。从Win32 Application Wizard到Application Settings，选择Static library，不选取Precompiled header，然后单击Finish。此时，你有了两个带创建配置信息的空项目Debug和Release，前者为活动配置。

接下来，在Solution Explorer中，右击项目名来显示项目属性页，并选择Properties。切换到Configuration Properties → Librarian → General，在标有Output File的字段中输入项目输出文件的路径名。路径名的目录应指向本章开始时生成的binaries目录，而文件名应为*libjohnpaul.lib*。

最后，使用Project菜单的Add Existing Item…把示例1-1的源文件添加到项目。现在，项目的属性页应含有一个标有“C/C++”的节点。切换到Configuration Properties → C/C++ → Code Generation，并指定Multi-threaded Debug DLL作为运行时库。现在，从Build菜单选择Build Solution就可以创建该项目了。你可以验证一下在*binaries*目录中已经生成了名为*libjohnpaul.lib*的文件。

不是用“Add Existing Item...”把示例1-1的源文件添加到项目中，而是用“Add New Item...”来生成空的源文件，再把这些空文件添加到项目中。然后，你就可以把示例1-1的内容从键盘输入或粘贴到新建文件中。

www.TopSage.com

讨论

不同IDE之间的差别比不同工具集之间的差别更大。每种IDE有自己的方式来生成项目、指定其配置属性以及往项目添加文件。但是，当你学会了使用几种 IDE 后，再学习使用其他 IDE 就容易了。

当学习使用一种新的 IDE 时，应重点注意一些特性：

- 如何生成一个新项目。
- 如何指定项目的类型（即可执行程序、静态库还是动态库）。
- 如何把已有的文件添加到项目中。
- 如何生成新文件并把它们添加到项目中。
- 如何指定项目输出文件的名称。
- 如何指定包含路径。
- 如何指定库查找路径。
- 如何指定项目所依赖的库。
- 如何创建项目。
- 如何把项目组织成组，并指定它们之间的依赖关系。

本技巧演示了其中很多的特性。其他特性在 1.12 节和 1.13 节中介绍。

下面让我们来看看如何使用 CodeWarrior、C++Builder 和 Dev-C++ 创建静态库。

CodeWarrior

从 File 菜单选择 New...，并选取 New 对话框的 Project 标签。输入 `libjohnpaul.mcp` 作为项目名，选择保存项目配置文件的位置，并双击 Mac OS C++ Stationery。从 New Project对话框中展开节点Mac OS X Mach-O和Standard Console，然后双击C++ Console Mach-O。此时，应有一个项目，它含有两个目标 Mach-O C++ Console Debug 和 Mach-O C++ Console Final，前者为默认目标。

当生成项目时，由于需要用名称来引用这些目标，所以应为这些目标给定描述名称。这里只是按如下步骤为调试目标进行了重命名。在项目窗口选择 Targets 标签，双击调试目标名，显示 Target Settings Window。然后切换到 Target → Target Settings，并在标有 Target Name 的字段中输入 `libjohnpaul Debug`。

接下来，从 Target Settings Window 切换到 Target → PPC Mac OS X Target。指定库作为项目类型，在标有 File Name 的字段中输入 `libjohnpaul.a`。切换到 Go to Target → Target Settings，并单击 Choose... 指定目录 *binaries* 作为输出文件 *libjpohnpaul.a* 的位置。

www.TopSage.com

最后，在项目窗口中选择 Files 标签，删除已有的源文件和库文件。然后使用 Project 菜单的 Add Files... 添加示例 1-1 的源文件。此时，可以选择 Project 菜单的 Make 来创建项目了。你可以验证一下在 binaries 目录中已经生成了名为 *libjohnpaul.a* 的文件。

C++Builder

从 File 菜单选择 New → Other…，然后选择 Library。此时就有了一个空项目。从 File 菜单选择 Save Project As…，选择一个保存项目配置文件的目录，并输入 *libjohnpaul.bpr* 作为项目名。

接下来，从 Project 菜单选择 Options…，显示 Project Options 对话框。然后，切换到 Directories and Conditionals，使用 Final output 旁的控制按钮指定项目输出文件 *libjohnpaul.lib* 位置。默认地，该文件将生成在 *libjohnpaul.bpr* 相同的目录中，但你应该告诉 C++Builder 生成在 *binaries* 目录中。如果你希望这样，也可以使用 *Intermediate output* 旁的控制按钮来指定目标文件的生成位置。默认地，它们也是生成在与源文件相同的目录中。

最后，使用 Project 菜单的 Add to Project…把示例 1-1 的源文件添加到项目中。通过选择 Project 菜单的 Make 就可以创建项目了。你可以验证一下在 *binaries* 目录中已经生成了名为 *libjohnpaul.lib* 的文件。

Dev-C++

选择 File 菜单的 New → Project…。从 New project 对话框中选择 Static Library 和 C++ Project，并输入 *libjohnpaul* 作为项目名。按下 OK 按钮后，指定保存项目配置文件的位置。

接下来，从 Project 菜单选择 Project Options，显示 Project Options 对话框。然后切换到 Build Options，验证项目输出文件的名字是 *libjohnpaul.a*。在可运行程序输出目录下输入 *binaries* 目录的路径名。

最后，使用 Project 菜单的 Add to project 把示例 1-1 的源文件添加到项目中。此时就可以选择 Execute 菜单的 Compile 来创建项目了。你可以验证一下在 *binaries* 目录中已经生成了名为 *libjohnpaul.a* 的文件。

参见 1.3 节、1.8 节和 1.16 节。

1.12 使用 IDE 创建动态库

问题

如何使用 IDE 从示例 1-2 的源文件创建动态库？

www.TopSage.com

大家网 TopSage.com

解决方案

基本轮廓如下：

1. 生成一个新项目，指定要创建动态库，而不是静态库或可执行程序。
2. 选择一个创建配置（如调试版或发布版，单线程或多线程）。
3. 指定库名和保存目录。
4. 把源文件添加到项目中。
5. 在 Windows 系统下，定义所需要的宏，确保动态库符号将使用 `__declspec(dllexport)` 来输出。
6. 如果有必要，指定一个或多个目录，告诉编译器到那里去查找所包含的头文件。见 1.13 节所示。
7. 创建项目。

正如 1.11 节所介绍的，该轮廓中的步骤可能会因不同的 IDE 而有所变化。例如，对于某些 IDE，其中有几步合成了一步，步骤的顺序也有所不同。第 2 步在 1.21 节、1.22 节和 1.23 节进行了详细介绍。这里，你尽可能使用默认设置。

例如，这里来看看如何使用 Visual C++ IDE 从示例 1-2 的源代码创建动态库。

从 File 菜单选择 New → Project，在左边面板中选择 Visual C++ Projects，选择 Win32 Console Application，并输入 *libgeorgeringo* 作为项目名。从 Win32 Application Wizard 到 Application Settings，选择 DLL 和 Empty Project，然后单击 Finish。此时，你有了两个带创建配置信息的空项目 Debug 和 Release，前者为活动配置。

接下来，在 Solution Explorer 中，右击项目名来显示项目属性页，并选择 Properties。切换到 Configuration Properties → Librarian → General，在标有 Output File 的字段中输入项目输出文件的路径名。路径名的目录应指向本章开始时生成的 binaries 目录，而文件名应为 *libgeorgeringo.dll*。同样，切换到 Configuration Properties → Linker → Advanced，在标有 Import Library 的字段中输入 DLL 输入库的路径名。路径名的目录部分应指向 *binaries* 目录，而文件名应为 lib*georgeringo.lib*。

接着，使用 Project 菜单的 Add Existing Item…把示例 1-2 的源文件添加到项目中。

不是用“Add Existing Item...”把示例 1-2 的源文件添加到项目中，而是用“Add New Item...”来生成空的源文件，再把这些空文件添加到项目中。然后，你就可以把示例 1-2 的内容从键盘输入或粘贴到新建文件中。

现在，项目的属性页应含有一个标有“C/C++”的节点。切换到 Configuration Properties → C/C++ → Code Generation，定义宏 `GEORGERINGO_DLL`，见 1.19 节。然后，切换到

www.TopSage.com

Configuration Propertiess → C/C++ → Code Generation，并指定 Multi-threaded Debug DLL 作为运行时库。

现在，从Build菜单选择Build Solution就可以创建该项目了。你可以验证一下在binaries目录中已经生成了名为 *libjohnpaul.lib* 的文件。

讨论

正如你在1.11节中所见到的那样，每个IDE都有自己的方式创建项目、指定项目配置属性以及添加文件到项目。下面我们来看看如何使用CodeWarrior、C++Builder和Dev-C++创建动态库。

CodeWarrior

从 File 菜单选择 New...，选择 New 对话框的 Project 标签，输入 **libgeorgeringo.mcp** 作为项目名，并选择保存项目配置文件的位置，双击 Mac OS C++ Stationery。从 New Project对话框中展开节点Mac OS X Mach-O和Standard Console，然后双击C++ Console Mach-O。此时，应有一个项目，它含有两个目标Mach-O C++ Console Debug 和 Mach-O C++ Console Final，前者为默认目标。

当生成项目时，由于需要用名称来引用这些目标，所以应为这些目标给定描述名称。这里只是按如下步骤为调试目标进行了重命名。在项目窗口选择Targets标签，双击调试目标名，显示 Target Settings Window。然后切换到 Target → Target Settings，并在标有 Target Name 的字段中输入 **libjohnpaul Debug**。

接下来，从 Target Settings Window 切换到 Target → PPC Mac OS X Target。指定动态库作为项目类型，在标有 File Name 的字段中输入 **libgeorgeringo.dylib**。切换到 Target → Target Settings，并单击 Choose... 指定目录 binaries 作为输出文件 *libgeorgeringo.dylib* 的位置。然后，切换到 Linker → PPC Mac OS X Linker。从标有 Export Symbols 的下拉列表框中选择 Use #pragma，确保标有 Main Entry Point 的字段为空。

最后，在项目窗口中选择 Files 标签，删除已有的源文件和库文件。然后使用 Project 菜单的 Add Files... 添加示例 1-2 的源文件。然后，使用 Add Files... 来添加目录 */usr/lib* 的文件 *dylib1.o*、目录 *Metrowerks CodeWarrior/MacOS X Support /Libraries /Runtime / Runtime_PPC/Runtime_MacOSX/Libs* 中的文件 *MSL_All_Mach-O_D.dylib* 和 *MSL_Shared_AppAndDylib_Runtime_D.lib*。如果要配置的是发布版，而不是调试版，则需要添加库 *MSL_All_Mach-O.dylib* 和 *MSL_Shared_AppAndDylib_Runtime.lib*。此时，可以选择 Project 菜单的 Make 来创建项目了。你可以验证一下在 *binaries* 目录中已经生成了名为 *libgeorgeringo.dylib* 的文件。

www.TopSage.com

C++Builder

从 File 菜单选择 New → Other…，然后选择 DLL Wizard。从 DLL Wizard 对话框中选择 C++ and Multi Threaded。此时，项目就有了单个文件 *Unit1.cpp*。右击并选择 Remove From Project 可以把文件 *Unit1.cpp* 从项目中删除掉。从 File 菜单选择 Save Project As…，选择一个保存项目配置文件的目录，并输入 *libgeorgeringo.bpr* 作为项目名。

接下来，从 Project 菜单选择 Options…，显示 Project Options 对话框。然后，切换到 Directories and Conditionals，使用 Final output 旁的控制按钮指定项目输出文件的位置。默认地，该文件将生成在 *libjohnpaul.bpr* 相同的目录中，但你应该告诉 C++Builder 生成在 *binaries* 目录中。如果你希望这样，也可以使用 Intermediate output 旁的控制按钮来指定目标文件的生成位置。默认地，它们也是生成在与源文件相同的目录中。

然后，定义宏 GEORGERINGO_DLL，见 1.19 节所示。

最后，使用 Project 菜单的 Add to Project…把示例 1-2 的源文件添加到项目中。通过选择 Project 菜单的 Make libgeorgeringo 就可以创建项目了。你可以验证一下在 binaries 目录中已经生成了名为 *libgeorgeringo.dll* 的文件。

Dev-C++

选择 File 菜单的 New → Project…。从 New project 对话框中选择 DLL and C++ Project，并输入 *libgeorgeringo* 作为项目名。按下 OK 按钮后，指定保存项目配置文件的位置。

接下来，从 Project 菜单选择 Project Options，显示 Project Options 对话框。然后切换到 Build Options，验证项目输出文件的名字是 *libjohnpaul.dll*。在可执行程序输出目录下输入 *binaries* 目录的路径名。

此时，定义宏 GEORGERINGO_DLL，见 1.19 节所示。

最后，通过右击把已有的文件从项目中删除掉。使用 File 菜单的 Save Project as…保存项目配置文件为 libgeorgeringo.dev。然后使用 Project 菜单的 Add to project 把示例 1-2 的源文件添加到项目中。此时就可以选择 Execute 菜单的 Compile 来创建项目了。你可以验证一下在 binaries 目录中已经生成了名为 libjohnpaul.a 的文件。

参见 1.4 节、1.9 节、1.17 节、1.19 节和 1.23 节。

1.13 使用 IDE 创建复杂应用程序

问题

如何使用 IDE 创建一个依赖多个静态库和动态库的可执行程序？

www.TopSage.com

解决方案

基本轮廓如下：

1. 如果要从源文件创建依赖库，这些库不是它们自己的IDE项目或makefiles所带有的，如1.11节和1.12节所介绍的那样创建它们。
2. 生成一个新项目，指定要创建一个可执行程序，而不是库。
3. 选择一个创建配置（例如，调试版或发布版，单线程或多线程）。
4. 指定可执行程序名，以及保存它的目录。
5. 把源文件添加到项目中。
6. 告诉编译器到哪里去查找所依赖库的头文件。
7. 告诉链接器使用了什么库以及到哪里去查找它们。
8. 如果IDE支持项目组，把上面的所有项目添加到单个项目组中，并指定它们之间的依赖关系。
9. 如果IDE支持项目组，从第8步中创建项目组。否则，就需要分别创建项目，注意，应先创建被其他项目依赖的项目。

如1.11节和1.12节一样，上面的步骤因不同的IDE而不同。第3步将在1.21节、1.22节和1.23节进行详细介绍。这里，你尽可能使用默认设置。

例如，这里来介绍如何使用Visual C++ IDE从示例1-3的源文件创建一个可执行程序。

选择File菜单的New → Project，选择左边面板的Visual C++，选择Win32 Console Application，并输入*hellobeatles*作为项目名。从Win32 Application Wizard到Application Settings，选择Console Application和Empty Project，并按下Finish按钮。此时，就有了一个空项目*hellobeatles.vcproj*，它有两个创建配置Debug和Release，前者为活动配置。你应该还有一个解决方案*hellobeatles.sln*，它含有单个项目*hellobeatles.vcproj*。

接下来，右击Solution Explorer中的项目名，显示项目的属性页，选择Properties。切换到Configuration Properties → Linker → General，并在标有Output File的字段中输入项目输出文件的路径名。其中，路径名的目录部分应指向binaries目录，文件名部分应为*hellobeatles.exe*。

然后，使用Project菜单的Add Existing Item…把示例1-3的源文件*helllobeatles.cpp*添加到项目中。此时，项目属性页应含有一个标识为C/C++的节点。切换到Configuration Properties → C/C++ → Code Generation，指定Multi-threaded Debug DLL作为运行时库。

www.TopSage.com

不是用“Add Existing Item...”把源文件hellbeatles.cpp添加到项目中，而是用“Add New Item...”生成一个空的*.cpp*源文件，并把它添加到项目中。然后，你就可以输入或粘贴示例 1-3 的内容到新建的文件中。

接着，切换到 Configuration Properties → C/C++ → General，在标有 Additional Include Directories的字段中输入包含有目录*johnpaul*和*georgeringo*的目录（即源文件john.hpp、ringo.hpp 等的父目录）。这使得头文件 *hellobeatles.hpp* 中的 `include` 指令可以正确解析。

使用 File 菜单的 Add → Existing Project…把项目文件 *libjohnpaul.vcproj* 和 *libgeorgeringo.vcproj* 添加到解决方案 hellobeatles。从 Project 菜单选择 Project Dependencies…，显示Project Dependencies对话框，从下拉控件中选择*hellobeatles*，并选中 *libjohnpaul* 和 *libgeorgringo* 旁的复选框。

注意： 如果要把多个项目添加到一个解决方案中，没有必要为每个项目生成一个解决方案。通过从File菜单中选择New → Blank Solution…来生成一个空的解决方案，然后从File菜单中选择 New → Project... 就可以把新项目添加到该解决方案中。

最后，从 Build 菜单中选择 Build Solution 来创建该解决方案。可以验证一下在目录 *binaries* 中生成了文件 *libjohnpaul.lib*、*libgeorgeringo.dll*、*libgeorgeringo.lib* 和 *hellobeatles.exe*。此时，从 Debug 菜单选择 Start Without Debugging 就可以运行该程序了。

讨论

在前面的示例中，很容易指定 *hellobeatles.exe* 依赖的库 *libjohnpaul.lib* 和 *libgeorgeringo.dll*，因为这两个库都是使用 Visual C++ 项目从源代码创建而来的。如果你创建的应用程序使用了带头文件的预创建的二进制文件，可以告诉 Visual C++ 如何查找它们。首先，切换到 Configuration Properties → C/C++ → General，在标有 Additional Include Directories 的编辑框中输入包含二进制头文件的目录。然后，切换到 Configuration Properties → Linker → Input，在标有 Additional dependencies 的字段中输入库名。最后，切换到Configuration Properties → Linker → General，在标有Additional Library Directories的编辑框中输入含有二进制文件的路径名。下面让我们来看看如何使用 CodeWarrior、C++Builder 和 Dev-C++ 从示例 1-3 的源文件创建可执行程序。

CodeWarrior

从File菜单选择New...，选择New对话框的Project标签，输入**`hellobeatles.mcp`**作为项目名，并选择保存项目配置文件的位置，双击Mac OS C++ Stationery。从New Project对话框中展开节点Mac OS X Mach-O和Standard Console，然后双击C++ Console Mach-

www.TopSage.com

大家网 TopSage.com

O。此时，应有一个项目，它含有两个目标Mach-O C++ Console Debug和Mach-O C++ Console Final，前者为默认目标。

由于需要用名称来引用这些目标，所以应为这些目标给定描述名称。这里只是按如下步骤为调试目标进行了重命名。在项目窗口选择Targets标签，双击调试目标名，显示Target Settings Window。然后切换到Target → Target Settings，并在标有Target Name的字段中输入**hellobeatles Debug**。

接下来，在项目窗口中选择Targets标签，双击调试目标名，显示Target Settings Window，切换到Target → PPC Mac OS X Target。指定可执行程序作为项目类型，在标有File Name的字段中输入**hellobeatles**。切换到Target→Target Settings，并单击Choose...指定目录*binaries*作为输出文件*libgeorgeringo.dylib*的位置。然后，切换到Linker→PPC Mac OS X Linker。从标有Export Symbols的下拉列表框中选择Use #pragma，确保标有Main Entry Point的字段为空。

在项目窗口中选择Files标签，删除已有的源文件和库文件。然后使用Project菜单的Add Files...把示例1-3的源文件hellobeatles.cpp添加到项目中。然后，使用Add Files...来添加目录*Metrowerks CodeWarrior/MacOS X Support /Libraries /Runtime /Runtime_PPC /Runtime_MacOSX /Libs*的文件*MSL_All_Mach-O_D.dylib*和*MSL_Shared_AppAndDylib_Runtime_D.lib*。如果要配置的是发布版，而不是调试版，则需要添加库*MSL_All_Mach-O.dylib*和*MSL_Shared_AppAndDylib_Runtime.lib*。从Target Settings Window，切换到Target → Access Paths，单击标有User Paths的面板。使用标有Add...按钮添加含有目录*johnpaul*和*georgeringo*的目录(即源文件*john.hpp*、*ringo.hpp*等的父目录)。这使得头文件*hellobeatles.hpp*中的include指令可以正确解析。

使用Project菜单的Add Files...，把项目文件*libjohnpaul.mcp*和*libgeorgeringo.mcp*添加到项目*hellobeatles.mcp*中。切换到Targets标签，扩展标有hellobeatles Debug、libjohnpaul.mcp和libgeorgeringo.mcp的节点。单击libjohnpaul.mcp和libgeorgeringo.mcp的子节点libjohgnpaul Debug和libgeorgeringo Debug目标图标。在这两个图标上将显示粗体箭头。如果有必要，可以放大你的项目窗口，展示窗口右边的小链接图标。在该列中单击两次，使目标图标与箭头对正。此时在该列中将出现两个黑点。

从Project菜单选择Make就可以创建该解决方案。链接器可能会显示一些关于多重定义符号的警告，这完全可以忽略不管。你也可以通过Linker → Mac OS X Linker并选中Suppress Warning Messages，不让这些警告出现。

请验证一下在binaries目录中已经生成了名为*libjohnpaul.a*、*libgeorgeringo.dylib*和*hellobeatles*的文件。此时，把库MSL_All_Mach-O_D.dylib复制到binaries目录中，并切换到*binaries*目录，在命令行中输入**./hellobeatles**就可以运行hellobeatles程序了。

www.TopSage.com

C++Builder

从File菜单选择New，然后选择Console Wizard。在Console Wizard对话框中，选择C++、Multi Threaded和Console Application。此时就有了含有单个源文件*Unit1.cpp*的项目，右击并选择Remove From Project可以把*Unit1.cpp*从项目中删除掉。从File菜单选择Save Project As…，选择用于保存项目配置文件的目录，并输入项目名*hello_beatles*。这里使用了一根下划线，因为C++ Builder不允许项目与源文件同名。

接下来，从Project菜单选择Options...，显示Project Options对话框。然后切换到Directories and Conditionals，使用Final output旁的按钮指定项目要生成的输出文件*hello_beatles.exe*。默认情况下，该文件将生成在与*hello_beatles.bpr*相同的目录中。可以告诉C++Builder把它生成在*binaries*目录中。你还可以使用Intermediate output旁的按钮来指定要生成的目标文件。默认地，它们也是生成在与源文件相同的目录中。

然后，使用Project菜单的Add to Project…，把示例1-3的源文件*helllobeatles.cpp*添加到项目中。

接着，从Project Options切换到Directories and Conditionals，使用Include path旁的按钮选择含有目录*johnpaul*和*georgeringo*的目录（即源文件*john.hpp*、*ringo.hpp*等的父目录）这使得头文件*hellobeatles.hpp*中的`include`指令可以正确解析。

右击标签ProjectGroup1，选择Save Project Group As，选择含有文件*hello_beatles.bpr*的目录，输入项目名*hello_beatles.bpg*。

通过右击标签hello_beatles并选择Add Existing Project…，可以把项目文件*libjohnpaul.bpr*和*libgeorgeringo.bpr*添加到你的项目组中。如果还没有创建这两个项目，可以按1.11节和1.12节所介绍的那样进行创建，然后利用Project菜单的Add to Project…把输出文件*libjohnpaul.lib*和*libgeorgeringo.lib*添加到项目hello_beatles。在Project Manager中，同时按下Ctrl和向上箭头键，把项目*libjohnpaul*和*libgeorgeringo*移到项目hello_beatles的上面，以确保首先创建这两个项目。

最后，从Build菜单选择Make All Projects来创建该解决方案。你可以验证一下在目录*binaries*中已经输出了名为*hellobeatles.exe*的文件。从Run菜单选择Run就可以运行该应用程序了。

Dev-C++

从File菜单选择New→Project…。在New project对话框中，选择Console Application和C++ Project，并输入*hellobeatles*作为项目名。按下OK按钮后，指定保存项目配置文件的位置。

接着，从Project Options切换到Build Options，验证一下项目输出文件是命名为

www.TopSage.com

hellobeatles.exe 了。在 Executable output 下输入目录 *binaries* 的路径名。你也可以在 Object file output director 下输入目录。

右击并选择 Remove file 可以把已有文件从项目中删除。使用 File 菜单的 Save Project as…可以保存项目配置文件 *hellobeatles.dev*。最后，使用 Project 菜单的 Add to project 把示例 1-3 的源文件添加到项目中。

然后，从 Project 菜单选择 Project Options，显示 Project Options 对话框。切换到 Directories→Include Directories，选择含有目录 *johnpaul* 和 *georgeringo* 的目录（即源文件 *john.hpp*、*ringo.hpp* 等的父目录），按下 Add 按钮。这使得头文件 *hellobeatles.hpp* 中的 include 指令可以正确解析。

最后，从 Project Options 中切换到 Directories→ Libraries Directories，添加将包含项目 *libjohnpaul* 和 *libgeorgeringo* 的输出文件 *libjohnpaul.a* 和 *libgeorgeringo.a* 的目录。然后切换到 Parameters→Linker，输入选项 *-ljohnpaul* 和 *-lgeorgeringo*。

此时，使用 Execute 菜单的 Compile 分别创建这三个项目，应确保最后创建 *hellobeatles*。选择 Execute 菜单的 Run 可以运行 *hellobeatles.exe*。

参见 1.5 节、1.10 节和 1.18 节。

1.14 获得 GNU make

问题

如何获得和安装 GNU make 实用工具，用于从源代码创建库和可执行程序？

解决方案

该解决方案取决于你的操作系统。

Windows

如果你从多个地方可以获得预创建的 GNU *make* 二进制文件，使 GNU *make* 的安装作为类 Unix 环境的一部分，我推荐使用 Cygwin 或 MSYS，它是 MinGW 项目的一部分。

注意：有关 Cygwin 和 MinGW 的介绍见 1.1 节。

如果你如 1.1 节所示那样安装了 Cygwin，那么你就已经有了 GNU *make*。要从 Cygwin 外壳程序运行它，只要运行命令 *make* 即可。

要安装 MSYS，先如 1.1 节所介绍的那样安装 MinGW。MinGW 安装程序的未来版本也许能让你自动安装 MSYS。但现在，需要遵循这些步骤。

www.TopSage.com

首先，到MinGW主页*http://www.mingw.org*，找到MinGW下载区域，下载MSYS安装程序的最新稳定版本。安装程序的名称应该为*MSYS-<version>.exe*。

接下来，运行安装程序。将会询问你指定MinGW安装的位置，MSYS应该安装的位置。当安装程序完成后，MSYS安装目录应该含有一个名为*msys.bat*的文件。运行这个脚本程序，将显示MSYS外壳程序，它是*bash*外壳程序的一部分，从这里你可以运行GNU *make*和其他mingw程序，如*g++*、*ar*、*ranlib*和*dlltool*。

注意：要使用MSYS，没有必要让MinGW安装程序或MSYS安装程序的*bin*子目录在PATH环境变量中。

Unix

首先，通过从命令行运行*make*，可以检查是否已安装了GNU *make*。如果GNU *make*已经安装了，将显示如下的消息：

```
GNU Make 3.80
Copyright (C) 2002  Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
...
```

如果你的系统有非GNU版本的*make*，可以把GNU版安装在*name gmake*下面。从命令行输入*gmake*就可以检验它。

如果你使用的是Mac OS X，获得GNU *make*最容易的方法是从Apple公司的网站下载Xcode开发环境，并按照指令进行安装。现在Xcode可以从*developer.apple.com/tools*下载。

否则，从*ftp://ftp.gnu.org/pub/gnu/make*下载GNU *make*的最新版本，解包并按照指令进行安装。

讨论

*make*实用工具有不少优点。大多数工具集都提供了*make*的版本。例如Visual C++有一个make实用工具名为*nmake.exe*。通常，特定工具集的make有一些内置的特性，使得它们能很容易与工具集一起使用。因此，关于各种工具集的*make*实用程序的讨论，要么得介绍几种版本的*make*，要么介绍一些案例，其中某个版本的*make*与工具集协调不是太好。

这里主要介绍的是GNU *make*，而不是多种*make*实用工具。GNU *make*是容易使用，功能最强大、可移植性好的*make*变种。最初，GNU *make*是设计用来与GCC一起使用的，因此，当GNU与其他工具集（特别是与Windows工具集）一起使用时，很需要技巧。但是，由于GNU *make*使用非常灵活，GNU与非GNU工具集一起使用比其他大多数make（如*make.exe*）要容易得多。

www.TopSage.com

GNU *make*功能强大，主要在于它可以运行复杂的外壳脚本程序。如果你要同时使用Unix和Windows，你就知道，Windows外壳程序*cmd.exe*不是很理想。它丢失了很多有用的命令，运行脚本程序的能力有限，命令行长度非常有限。因此，让GNU *make*去使用*cmd.exe*大大地限制了它的短处。但是，Cygwin和MSYS为Windows系统下的GNU *make*的使用提供了很好的环境。

MSYS提供了运行Unix风格的makefile和Windows系统下的配置脚本程序所需要的最小环境。它提供的有用的工具包括*awk*、*cat*、*cp*、*grep*、*ls*、*mkdir*、*mv*、*rm*、*rmdir*和*sed*。MSYS设计为与GCC一起使用，它也的确做得很漂亮，但是，它与其他的Windows工具集一起使用有时候不是很顺畅，尤其是那些用*.bat*文件作为设置环境，使用斜杠（/）而不是连字符（-）作为命令行选项的工具集。

MSYS是最小化的，而Cygwin则是最大化的。Cygwin *make*可以完成MSYS *make*所能做的全部工作，而且还多得多。但是，可移植的makefile文件限制它们自己只能用于GNU工具，而MSYS则可以支持全部。

参见1.1节。

1.15 使用GNU make创建“Hello, World”应用程序

问题

如何用GNU make从示例1-4创建简单的“Hello, World”程序？

解决方案

在你编写你的第一个makefile文件之前，需要了解一些术语。一个makefile文件由如下形式的规则集组成：

```
targets: prerequisites
    command-script
```

这里，`targets`和`prerequisites`是用空格隔开的字符串，命令脚本由0行或多行文本组成，每行以一个制表符开头。targets和prerequisites通常是文件名，但有时候它们也可以是*make*要执行的动作。命令脚本由一系列要传递给外壳程序的命令组成。粗略地说，就是规则告诉*make*，通过运行命令脚本，从prerequisites生成targets集。

注意：makefile文件中的空格是有意义的。含有命令脚本的行必须以制表符而不是空格打头（这是初学者最易犯错误的地方）。在下面的示例中，以制表符打头的行用四个字符来表示。

现在你就可以开始了。在含有源文件的目录中生成一个名为*makefile*的文本文件。在这个文件中，声明四个目标。调用第一个目标`all`，并把可执行程序名指定为唯一的先决

www.TopSage.com

大家网 TopSage.com

条件。它应该还没有命令脚本。给第二个目标提供与可执行程序相同的名字。把应用程序的源文件指定为它的先决条件，并把用于从源文件创建可执行程序的命令行指定为目标的命令脚本。第三个目标应调用 install。它没有先决条件，但有一个命令脚本，用于从含有 makefile 的目录中把可执行程序复制到要安装它的目录中。最后一个目标调用 clean。与 install 一样，它也没有先决条件。其命令脚本用于从当前目录中删除可执行程序和中间目标文件。clean 和 install 目标应使用 PHONY 属性标识为 *phony* 目标。

例如，要使用 GCC 把示例 1-4 的源文件创建为可执行程序，其 makefile 文件应如示例 1-14 所示。

示例 1-14 用 GCC 创建可执行程序的 makefile 文件

```
# This is the default target, which will be built when
# you invoke make
.PHONY: all
all: hello

# This rule tells make how to build hello from hello.cpp
hello: hello.cpp
    g++ -o hello hello.cpp

# This rule tells make to copy hello to the binaries subdirectory,
# creating it if necessary
.PHONY: install
install:
    mkdir -p binaries
    cp -p hello binaries

# This rule tells make to delete hello and hello.o
.PHONY: clean
clean:
    rm -f hello
```

要用 Visual C++ 把示例 1-4 的源代码创建成可执行程序，其 makefile 文件如示例 1-15 所示。

示例 1-15 用 Visual C++ 创建可执行程序的 makefile 文件

```
#default target
.PHONY: all
all: hello.exe

#rule to build hello.exe
hello.exe: hello.cpp
    cl -nologo -EHsc -GR -Zc:forScope -Zc:wchar_t \
        -Fehello hello.cpp

.PHONY: install
install:
    mkdir -p binaries
    cp -p hello.exe binaries

.PHONY: clean
clean:
    rm -f hello.exe
```

注意： 在 makefile 文件中，命令和目标或先决条件的列表用连接字符“\”可以占多行，就像在 C++ 源文件中一样。

www.TopSage.com

要创建可执行程序，设置好命令行工具所需的环境变量，切换到含有 *makefile* 文件的目录中，输入**make**目录。要把可执行程序复制到 *binaries* 目录中，输入**make install**目录。要从makefile目录中删除可执行程序和中间目标文件，输入**make clean**命令。

注意： 如果按1.1节所示的那样已经安装了Cygwin开发环境，就可以直接从Windows系统的外壳程序 *cmd.exe* 运行示例1-15的makefile文件。

注意： 你也可以如下从Cygwin运行这个makefile文件。从 *cmd.exe* 中，运行 *vcvars32.bat*，用来设置Visual C++的环境变量。接着，运行 *cygwin.bat* 来启动Cygwin外壳程序。如果已经把Cygwin的安装目录放置在了PATH中，就可以从 *cmd.exe* 中输入**cygwin**来启动Cygwin外壳程序。最后，切换到含有makefile文件的目录，输入**make**。

同样，也可以从MSYS外壳程序运行makefile文件：从 *cmd.exe* 运行 *vcvars32.bat*，然后运行 *msys.bat* 来启动MSYS外壳程序。

如果工具集提供了用于设置环境变量的脚本，从Cygwin或MSYS运行makefile文件比从 *cmd.exe* 中运行更棘手一些。但是，这对一些makefile文件是需要的，因为它们不能从 *cmd.exe* 运行。

讨论

在接下来的几个技巧中，你将看到，GNU *make* 是创建复杂项目的强大工具。但它实际是如何做的呢？这里来解释一下它的工作原理。当不带参数调用 *make* 时，它就像当前目录中名为 *GNUmakefile*、*makefile* 或 *Makefile* 的文件一样，试图创建它所包含的第一个目标，即调用默认的目标。如果默认目标是 *up to date* —— 即它已经存在，它所有的先决条件也是up to date，也就是说，所有先决条件都是最新的 —— *make* 的工作就完成了。否则，通过运行命令脚本，它将从先决条件生成默认目标。与 *up to date* 的定义一样，这个过程是反复进行的：对每个不是最新的先决条件，*make* 都将查找一个把先决条件作为目标的规则，并重新开始整个过程。这将不断进行下去，直到默认目标已经更新或发生了一个错误为止。

从上面的描述中可知，当且仅当目标没有对应的文件时，没有先决条件的目标才是最新的。因此，与不存在的文件对应的目标永远不是最新的，可以用来使命令脚本无条件地运行。这种目标就称为 *phony targets*。

注意： 通过在目标前面加.PHONY属性，如示例1-14和示例1-15所示，就可以告诉 *make*，该目标没有对应的文件，因此它总是要重新创建。

反过来，与已有文件对应的先决条件总是最新的，因此它不会作为规则的目标而出现。

www.TopSage.com

现在让我们来看看，当运行示例 1-14 的 makefile 文件时都发生了些什么。目标 all 总是要的：它的唯一目的就是告诉 make 去创建 *hello.exe*。在这个简单的 makefile 中，all 目标没有先决条件，在更复杂的示例中，all 目标可能需要多个先决条件。目标 hello 的规则告诉 *make*，如果需要，调用 *g++* 来创建 *hello*。假设在当前目录中只有 *makefile* 和 *hello.cpp*，目标 hello 不是最新的。但是，先决条件是最新的，因为文件 *hello.cpp* 已经存在，且 hello.cpp 没有作为目标出现在任何规则中。因此，make 调用 g++ 来编译和链接 *hello.cpp*，生成 *hello* 文件。all 目标的先决条件现在是最新的了，因此，*make* 创建 all 目标 —— 通过运行一个空命令脚本 —— 并退出。

当用对应一个目标的命令行参数来调用 make 时，make 将试图创建这个目标。因此，运行 **make install** 将导致下面命令的运行：

```
mkdir -p binaries
cp -p hello binaries
```

第一个命令生成目录 *binaries*（如果它不存在），第二个命令把 hello 复制到该目录中。同样，**make clean** 将调用以下命令：

```
rm -f hello
```

它将删除 *hello*。

注意：如果你使用的是 Windows 系统，install 和 clean 目标使用的 mkdir、cp 和 rm 命令指的是随 Cygwin 或 MSYS 发布的 GNU 工具。

一旦你了解了 *make* 的原理，示例 1-14 看起来就相对简单了。但是，实际上，它比它实际需要的还要复杂，看看简化它的不同方法，对学习 makefile 的初步知识是一种很好的方式。

make 变量

GNU *make* 支持值为字符串的变量。在 makefile 中使用最普遍的变量是符号常量。我们往往不是硬编码文件名或在 makefile 的多个地方使用某个外壳程序命令，而是把文件或命令名赋给一个变量，从而可以使用该变量。这使得维护 makefile 文件更简单也更容易。例如，使用变量重新编写示例 1-14 的 makefile 文件，如示例 1-16 所示。

示例 1-16 用 GCC 创建可执行程序 hello 的 makefile 文件，使用 make 变量

```
# Specify the target file and the install directory
OUTPUTFILE=hello
INSTALLDIR=binaries

# Default target
.PHONY: all
all: $(OUTPUTFILE)
```

www.TopSage.com

```
# Build hello from hello.cpp
$(OUTPUTFILE): hello.cpp
    g++ -o hello hello.cpp

# Copy hello to the binaries subdirectory
.PHONY: install
install:
    mkdir -p $(INSTALLDIR)
    cp -p $(OUTPUTFILE) $(INSTALLDIR)

# Delete hello
.PHONY: clean
clean:
    rm -f $(OUTPUTFILE)
```

这里引入了两个make变量OUTPUTFILE和INSTALLDIR。正如你所看到的那样，可以使用赋值运算符=给make变量赋值，通过用括号把它们括起来并加上美元符号作为前缀，就可以对它们进行计算。

使用语法*make X=Y*，就可以在命令行中设置make变量的值。而且，当启动make时，每个环境变量可用于初始化同名和同值的*make*变量。在命令行中指定的值比从环境中继承来的值有更高的优先级，在makefile指定的值又比在命令行中指定的值有更高的优先级。

GNU make还支持自动变量，当在命令脚本中计算时，赋给特定的值。最重要的是，变量$@表示的是目标文件名，变量$<表示的是第一个先决条件的文件名，而变量$^表示的是用空格隔开的先决条件系列。使用这些变量，我们就可以进一步简化示例1-16的makefile文件了，如示例1-17所示。

示例1-17 用GCC创建可执行程序hello的makefile文件，使用自动变量

```
# Specify the target file and the install directory
OUTPUTFILE=hello
INSTALLDIR=binaries

# Default target
.PHONY: all
all: $(OUTPUTFILE)

# Build hello from hello.cpp
$(OUTPUTFILE): hello.cpp
    g++ -o $@ $<

# Install and clean targets as in 示例1-16
```

在命令脚本g++ -o $@ $<中，变量$@扩展为hello，而变量$<扩展为hello.cpp。因此，示例1-17的makefile文件等同于示例1-16的，但代码的重复性降低了。

隐式规则

示例1-17的makefile还可以进一步简化。实际上，与hello目标相关的命令脚本是多余的，你可以使用如示例1-18所示的makefile文件。

示例1-18 用GCC创建可执行程序hello的makefile文件，使用隐式规则

```
# Specify the target file and the install directory
```

www.TopSage.com

```
OUTPUTFILE=hello
INSTALLDIR=binaries

# Default target
.PHONY: all
all: $(OUTPUTFILE)

# Tell make to rebuild hello whenever hello.cpp is modified
$(OUTPUTFILE): hello.cpp

# Install and clean targets as in 示例1-16
```

make是如何知道从源文件*hello.cpp*创建可运行程序*hello*的呢？答案是*make*维护一个隐式规则的内部数据库，表示当创建C和C++应用程序时通常执行的操作。例如，从.cpp文件生成可执行程序的隐式规则如示例1-19所示。

示例1-19 make的内部数据库的模式规则

```
%: %.cpp
#  commands to execute (built-in):
    $(LINK.cpp) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

第一行具有*%xxx:%yyy*形式的规则称为模式规则，字符%起到一个通配符的作用。当没有规则与过时的先决条件匹配时，make将查找可用的模式规则。对每种模式规则，*make*将试图查找一个字符串，该字符串替代模式规则中的目标的通配符，从而产生一个过时的先决条件。如果*make*找到了这样一个字符串，*make*将替代模式规则中目标和先决条件的通配符，以生成一个新规则。然后，make试图用这个新规则来创建过时的先决条件。

注意：可以使用make -p来显示GNU make的隐式规则数据库。

例如，当首次运行示例1-18的makefile时，默认目标`all`的先决条件`hello`是过时的。尽管在规则`$(OUTPUTFILE):hello.cpp`中hello是作为一个目标而出现的，但这个规则没有命令脚本，因此对创建文件*hello.make*无用，从而要查找其内部数据库，找到示例1-19所示的规则。通过用hell。把示例1-19的规则的通配符替换掉，*make*就可以生成以下规则，其中`hello`作为它的目标了：

```
hello: hello.cpp
    $(LINK.cpp) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

到此为止是很好的——但很显然，还有好多要介绍的。再来看看*make*的内部数据库，可以发现，默认地，变量`LINK.cpp`扩展为`$(LINK.cc)`了。反过来，默认地，`LINK.cc`扩展为：

```
$(CXX) $(CXXFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)
```

最后，默认地，变量CXX扩展为`g++`，其他四个变量`$(CXXFLAGS)`、`$(CPPFLAGS)`、`$(LDFLAGS)`和`$(TARGET_ARCH)`扩展为空字符串。当执行完所有这些替代后，就剩下如下规则了，这对我们很熟悉：

www.TopSage.com

大家网 TopSage.com

```
hello: hello.cpp
    g++ $^ -o $@
```

注意： 是不是很困惑了？这不能怪你。如果你学习上面的解释，并花些时间来查看*make*的内部数据库，就会对隐式规则开始有所了解了。

自定义部分

现在你知道了示例 1-19 的模式规则是如何让 *make* 从源文件 *hello.cpp* 创建可执行程序 *hello*的了，你可能还想知道，为什么需要这么多的中间步骤？为什么不添加下面的规则：

```
%: %.cpp
    g++ $^ -o $@
```

来查看*make*的内部数据库，而不是使用示例1-19所示的复杂规则？答案是诸如$(CXX)、$(CXXFLAGS)、$(CPPFLAGS)和$(LDFLAGS)之类的中间变量起着自定义的作用。例如，你可以在命令行通过为makefile文件的LDFLAGS指定一个值，或者通过设置环境变量，把其余的标识传递给链接器。变量CPPFLAGS和CXXFLAGS所起的作用分别与C++的预处理器或编译器选项类似。你甚至可以通过设置变量CXX而不是GCC来指定编译器。例如，要利用示例 1-18 的 makefile 文件，使用 Intel for Linux 来创建 *hello* 程序，可以从命令行输入 **`make CXX=icpc`**，前提是你已经设置了 Intel 编译器所需的环境变量。

VPATH 与 vpath 指令

在示例 1-18 中，可以用 *make* 来纠正模式规则，因为 *.cpp* 文件与要生成的输出文件在同一个目录中。如果源文件在不同的目录中，你可以使用 VPATH 变量，告诉 make 到哪里去查找目标或先决条件：

```
VPATH = <path-to-cpp-files>
```

你也可以使用 vpath 指令来告诉 *make* 在某个位置中查找某些类型的文件：

```
# look for .exp files in ../lib
vpath %. exp../lib
```

参加 1.2 节和 1.7 节。

1.16 使用 GNU make 创建静态库

问题

如何用 GNU *make* 从示例 1-1 的源文件创建静态库？

解决方案

首先，在要生成静态库的目录中生成一个makefile文件，并把只有单个先决条件的phony目标声明为静态库。接着，声明静态库目标。其先决条件应为该库所包含的目标文件，其

www.TopSage.com

命令应为用于从目标文件集合创建库的命令行，如 1.3 节所示。如果你使用的是 GCC 或有类似命令行语法的编译器，如果有必要，可以通过修改在 *make* 的数据库的隐式规则中使用的参数 CXX、CXXFLAGS 等来定制隐式的模式规则，如 1.15 节所示。否则，就需要编写一个模式规则，使用表 1-8 所示的命令行和 1.16 节介绍的模式规则语法，告诉 *make* 如何把 *.cpp* 文件编译成目标文件。然后，声明目标，说明每个库的源文件如何直接或间接地依赖于它所包含的头文件。你可以手工编写以及让它们自动生成这些依赖关系。最后，添加 install 和 clean 目标，如 1.15 节所示。

例如，要在 Unix 系统下使用 GCC 从示例 1-2 的源文件创建静态库，需要在 *johnpaul* 目录中生成一个，如示例 1-20 所示。

示例 1-20 Unix 系统下使用 GCC 创建 ibjohnpaul.a 的 makefile 文件

```
# Specify extensions of files to delete when cleaning
CLEANEXTS   = o a

# Specify the target file and the install directory
OUTPUTFILE  = libjohnpaul.a
INSTALLDIR  = ../binaries

# Default target
.PHONY: all
all: $(OUTPUTFILE)

# Build libjohnpaul.a from john.o, paul.o, and johnpaul.o
$(OUTPUTFILE): john.o paul.o johnpaul.o
    ar ru $@ $^
    ranlib $@

# No rule to build john.o, paul.o, and johnpaul.o from .cpp
# files is required; this is handled by make's database of
# implicit rules

.PHONY: install
install:
    mkdir -p $(INSTALLDIR)
    cp -p $(OUTPUTFILE) $(INSTALLDIR)

.PHONY: clean
clean:
    for file in $(CLEANEXTS); do rm -f *.$$file; done

# Indicate dependencies of .ccp files on .hpp files
john.o: john.hpp
paul.o: paul.hpp
johnpaul.o: john.hpp paul.hpp johnpaul.hpp
```

同样，要使用 Visual C++ 创建静态库，makefile 文件如示例 1-21 所示。

示例 1-21 使用 Visual C++ 创建静态库 libjohnpaul.lib 的 makefile 文件

```
# Specify extensions of files to delete when cleaning
CLEANEXTS      = obj lib

# Specify the target file and the install directory
OUTPUTFILE     = libjohnpaul.lib
INSTALLDIR     = ../binaries
```

www.TopSage.com

```
# Pattern rule to build an object file from a .cpp file
%.obj: %.cpp
    "$(MSVCDIR)/bin/cl" -c -nologo -EHsc -GR -Zc:forScope -Zc:wchar_t \
            $(CXXFLAGS) $(CPPFLAGS) -Fo"$@" $<

# Default target
.PHONY: all
all: $(OUTPUTFILE)

# Build libjohnpaul.lib from john. obj, paul. obj, and johnpaul. obj
$(OUTPUTFILE): john.obj paul.obj johnpaul.obj
    "$(MSVCDIR)/bin/link" -lib -nologo -out:"$@" $^

.PHONY: install
install:
    mkdir -p $(INSTALLDIR)
    cp -p $(OUTPUTFILE) $(INSTALLDIR)

.PHONY: clean
clean:
    for file in $(CLEANEXTS); do rm -f *.$$file; done

# Indicate dependency of .cpp files on .hpp files
john.obj: john.hpp
paul.obj: paul.hpp
johnpaul. obj: john.hpp paul.hpp johnpaul.hpp
```

注意： 在示例1-21中，我把Visual C++的*link.exe*命令写成了`"$(MSVCDIR)/bin/link"`，通过*vcvars32.bat*来设置环境变量MSVCDIR。这可以避免使Cygwin和MSYS支持的Visual C++链接器和Unix的*link*命令发生混淆。为了简洁起见，我还用MSVCDIR来表示Visual C++的编译命令。

讨论

让我们来浏览一下示例1-20。我开始定义了用来表示输出文件、安装目录和文件扩展名的变量，这些在安装完成后将被删除。接下来，我声明了默认目标`all`，如示例1-14所示。

用于创建静态库的规则如下所示：

```
$(OUTPUTFILE): john.o paul.o johnpaul.o
    ar ru $@ $^
    ranlib $@
```

这里直接采用了表1-10的GCC项。其中，`$(OUTPUTFILE)`和`$@`扩展为`libjohnpaul.a`，`$^`扩展为先决条件列表`john.o paul.o johnpaul.o`。

接下来的两行声明`install`和`clean`目标，如1.15节所示。唯一不同的是，在示例1-20中使用了一个外壳循环结构，用来删除扩展名为`o a`等的所有文件以及所有目标文件或静态库文件。

```
for file in $(CLEANEXTS); do rm -f *.$$file; done
```

这里使用了两个美元符号，防止*make*扩展变量`$$file`，而不是把它传递给外壳程序。

www.TopSage.com

最后三行指定了库的*.cpp*文件与它们所包含的头文件之间的依赖关系。每个规则对应一个*.cpp*文件，其目标是要从*.cpp*文件创建的目标文件，其先决条件是由*.cpp*文件间接或直接包含的头文件：

```
john.o: john.hpp
paul.o: paul.hpp
johnpaul.o: john.hpp paul.hpp johnpaul.hpp
```

这可以如下来理解。如果一个*.cpp*文件间接或直接地包含了一个头文件，每当头文件被修改后，都要重新创建。但是，由于*.cpp*文件存在且没有作为任何规则的目标而出现，它就永远不会过时，见 1.15 节。接下来，当头文件被修改了，不会触发重新编译的命令。保险的做法是声明一个规则，使这些依赖关系为显式的，只要修改了一个头文件，对应于该*.cpp*的目标文件就是过时的，从而使得*.cpp*文件重新编译。

这种解决方案只适合于很小型的项目，因为要使表示源文件的目标与发生改变的代码保持同步是非常困难的。幸运的是，有几种用于自动生成这种依赖关系的方法。例如，可以使用如下内容来替换示例 1-20 的最后三行：

```
# Generate dependencies of .cpp files on .hpp files
include john.d paul.d johnpaul.d

%.d: %.cpp
    $(CC) -M $(CPPFLAGS) $< > $@.$$$$; \
    sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
    rm -f $@.$$$$
```

上面代码是依靠编译器选项-M来使得GCC输出依赖信息。关于它的工作原理的详细介绍，请参见*Managing Projects with GNU make*, Third Edition一书，作者为Robert Mecklenburg，O'Reilly出版社出版。

注意：在makefile文件的末尾，需要放置用于生成依赖关系的代码。

该方法可用来与大多数工具集一起使用，因为大多数的编译器都提供了一个类似于GCC的*-M*的选项。事实上，该选项通常为*-M*或*-m*。但是，Visual C++不提供用于生成makefile依赖关系的选项。如果你使用的是Visual C++，有两种选择。你可以使用*-Gm*选项，与*-Zi*或*-ZI*选项一起使用，详细讨论见1.21节。*-Gm*选项告诉编译器创建一个数据库，保存在扩展名为*idb*的文件中，它包含了有关源文件之间的相互关系的信息。当一个或多个*.cpp*文件首次编译时，都会生成这个*.idb*文件。在一系列的编译工作中，只重新编译那些本身被修改过的源文件，或其所依赖的头文件被修改过的源文件。

你也可以同时使用*-showIncludes*和*-E*选项。每当遇到一个`include`指令时，*-showIncludes*选项都会使编译器往标准错误输出一个消息。*-E*选项告诉编译器，运行预处理器，然后退出，不创建任何二进制文件。使用一些外壳脚本，就可以使用由*-showIncludes*选项生成的输出来构建makefile的依赖关系：

```
include john.d paul.d johnpaul.d
```

www.TopSage.com

```
%.d: %.cpp
     "$(MSVCDIR)/bin/cl" -E -showIncludes $< 2> $@.$$$$ > /dev/null; \
     sed -n 's/^Note: including file: *\(.*\)/$*.obj$*.d:\1/gp' \
        < $@.$$$$ | sed 's:\\:/:g;s: :\\ :gp' > $@;              \
     rm -f $@.$$$$
```

在这个示例中，字符•代表一个制表符。

让我们来对示例1-20做最后一个改进。目前，序列john paul johnpaul在两个地方出现了：在创建静态库的规则的先决条件中，以及在用于生成依赖关系的include指令中。如果源文件列表发生改变，你就需要在这两个地方更新makefile文件。最好是定义一个变量SOURCES，并用含有SOURCES的表达式替换这两处的john paul johnpaul序列：

```
SOURCES = john.cpp paul.cpp johnpaul.cpp
...
# Build libjohnpaul.a from john.o, paul.o, and johnpaul.o
$(OUTPUTFILE): $(subst .cpp,.o,$(SOURCES))
    ar ru $@ $^
    ranlib $@
...

# Generate dependencies of .ccp files on .hpp files
include $(subst .cpp,.d,$(SOURCES))

%.d: %.cpp
     $(CC) -M $(CPPFLAGS) $< > $@.$$$$; \
     sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
     rm -f $@.$$$$
```

这里使用了*make*函数`$(subst x,y,str)`，它把`str`中的所有`x`用`y`进行替换。

注意： GNU make支持大量的字符串函数和文件操作函数。它还支持自定义函数。同样，有关这方面的详细介绍，请参见*Managing Projects with GNU make*, Third Edition一书（Robert Mecklenburg, O'Reilly）。

参加1.2节和1.7节。

1.17 使用GNU make创建动态库

问题

如何使用GNU *make*从示例1-2的C++源文件创建动态库？

解决方案

首先，在要生成动态库的目录中生成一个makefile文件，并把只有单个先决条件的phony目标声明为动态库。接着，声明动态库目标。其先决条件应为用于创建该库的目标文件，其命令应为用于从目标文件集合创建该库的命令行，如1.4节所示。如果你使用的是GCC或有类似命令行语法的编译器，如果有必要，可以通过修改在*make*的数据库的隐式规则中使用的参数CXX、CXXFLAGS等来定制隐式的模式规则，如1.15节所示。否则，就需要编写一个模式规则，使用表1-8所示的命令行和1.16节介绍的模式规则语法，告诉*make*

www.TopSage.com

大家网 TopSage.com

如何把 *.cpp* 文件编译成目标文件。最后，添加 `install` 和 `clean` 目标，如 1.15 节所示，以及添加自动生成源文件依赖关系的机制，如 1.16 节所示。

例如，要使用 GNU make 从示例 1-2 的 C++ 源文件创建动态库，在 *georgeringo* 目录中生成一个 makefile 文件，如示例 1-22 所示。

示例 1-22 用 GCC 创建 libgeorgeringo.so 的 makefile 文件

```
# Specify extensions of files to delete when cleaning
CLEANEXTS   = o so

# Specify the source files, the target files,
# and the install directory
SOURCES     = george.cpp ringo.cpp georgeringo.cpp
OUTPUTFILE  = libgeorgeringo.so
INSTALLDIR  = ../binaries

.PHONY: all
all: $(OUTPUTFILE)

# Build libgeorgeringo.so from george.o, ringo.o,
# and georgeringo.o; subst is the search-and-replace
# function demonstrated in Recipe 1.16
$(OUTPUTFILE): $(subst .cpp,.o,$(SOURCES))
    $(CXX) -shared -fPIC $(LDFLAGS) -o $@ $^

.PHONY: install
install:
    mkdir -p $(INSTALLDIR)
    cp -p $(OUTPUTFILE) $(INSTALLDIR)

.PHONY: clean
clean:
   for file in $(CLEANEXTS); do rm -f *.$$file; done

# Generate dependencies of .ccp files on .hpp files
include $(subst .cpp,.d,$(SOURCES))

%.d: %.cpp
    $(CC) -M $(CPPFLAGS) $< > $@.$$$$; \
    sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
     rm -f $@.$$$$
```

讨论

示例 1-22 的 makefile 是 1.4 节、1.15 节和 1.16 节的直接应用。示例 1-22 与示例 1-20 的最大不同是用于从目标文件 *george.o*、*ringo.o* 和 *georgeringo.o* 创建 *lingeorgeringo.so* 的规则：

```
$(OUTPUTFILE): $(subst .cpp,.o,$(SOURCES))
    $(CXX) -shared -fPIC $(LDFLAGS) -o $@ $^
```

这里 `$(OUTPUTFILE)` 扩展为 `lingeorgeringo.so`，表达式 `$(subst .cpp,.o,$(SOURCES))` 扩展为 `george.o ringo.o georgeringo.o`，如 1.16 节所示。命令脚本 `$(CXX) -shared -fPIC $(LDFLAGS) -o $@ $^` 采用的是表 1-11 所示的 GCC 命令行。

参见 1.4 节、1.9 节、1.12 节、1.19 节和 1.23 节。

www.TopSage.com

1.18 使用 GNU make 创建复杂应用程序

问题

如何用 GNU *make* 创建一个依赖于多个静态库和动态库的可执行程序?

解决方案

遵循步骤如下:

1. 为应用程序所使用的库创建 makefile 文件,见 1.16 节和 1.17 节。这些 makefile 文件必须位于不同的目录中。
2. 在另外一个目录中创建一个 makefile 文件。这个 makefile 可以用来创建应用程序,但必须是在运行了上面第 1 步之后。给定这个 makefile 文件为 phony target `all`,其先决条件是可执行程序。为可运行程序声明一个目标,其先决条件由应用程序所用的库组成,以及由应用程序的*.cpp* 文件创建目标文件。编写一个命令脚本程序来从集合库和目标文件创建可执行程序,如 1.5 节所示。如果有必要,也可以编写一个模式规则,用来从*.cpp* 文件生成目标文件,如 1.16 节所示。如 1.15 节所示的那样添加 `install` 和 `clean`,并如 1.16 节所示的那样自动生成源文件的依赖关系。
3. 在包含所有其他 makefile 文件的父目录中生成一个 makefile 文件 —— 假设我们称这个新的 makefile 文件为 *top-level*,其他的 makefile 文件为 *subordinate*。声明一个默认的 target `all`,其先决条件是含有在第 2 步中创建的 makefile 文件的目录。声明一个规则,其目标由含有所有 *subordinate* makefile 文件的目录组成,其命令脚本将激活每个目标目录中的 make,其中,目标被指定为变量 `TARGET` 的值。最后,声明指定 subordinate makefile 文件中默认目标的依赖关系。

例如,要使用 Unix 系统下的 GCC 从示例 1-3 的源文件创建一个可执行程序,可以生成示例 1-23 所示的 makefile 文件。

示例 1-23 用 GCC 创建 hellobeatles.exe 的 makefile 文件

```
# Specify the source files, target files, the build directories,
# and the install directory
SOURCES         = hellobeatles.cpp
OUTPUTFILE      = hellobeatles
LIBJOHNPAUL     = libjohnpaul.a
LIBGEORGERINGO = libgeorgeringo.so
JOHNPAULDIR     = ../johnpaul
GEORGERINGODIR     = ../georgeringo
INSTALLDIR      = ../binaries

#
# Add the parent directory as an include path
#
CPPFLAGS      += -I..

#
# Default target
#
```

www.TopSage.com

```
.PHONY: all
all: $(HELLOBEATLES)

#
# Target to build the executable.
#
$(OUTPUTFILE): $(subst .cpp,.o,$(SOURCES))  \
               $(JOHNPAULDIR)/$(LIBJOHNPAUL) \
               $(GEORGERINGODIR)/$(LIBGEORGERINGO)
    $(CXX) $(LDFLAGS) -o $@ $^

.PHONY: install
install:
    mkdir -p $(INSTALLDIR)
    cp -p $(OUTPUTFILE) $(INSTALLDIR)

.PHONY: clean
clean:
    rm -f *.o
    rm -f $(OUTPUTFILE)

# Generate dependencies of .ccp files on .hpp files
include $(subst .cpp,.d,$(SOURCES))

%.d: %.cpp
    $(CC) -M $(CPPFLAGS) $< > $@.$$$$; \
    sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
    rm -f $@.$$$$
```

接着，在含有*johnpaul*、*georgeringo*、*hellobeatles*和*binaries*的目录中生成一个高层的makefile文件，如示例1-24所示。

示例1-24 用于示例1-1～示例1-3的源文件的高层makefile文件

```
# All the targets in this makefile are phony
.PHONY: all johnpaul georgeringo hellobeatles

# Default target
all: hellobeatles

# The targets johnpaul, georgeringo, and hellobeatles represent
# directories; the command script invokes make in each directory
johnpaul georgeringo hellobeatles:
    $(MAKE) --directory=$@ $(TARGET)

# This rule indicates that the default target of the makefile
# in directory hellobeatles depends on the default targets of
# the makefiles in the directories johnpaul and georgeringo
.PHONY: hellobeatles
hellobeatles: johnpaul georgeringo
```

要创建hellobeatles，切换到含有高层makefile的目录，输入**make**命令。要把文件*libjohnpaul.a*、*libgeorgeringo.so*和*hellobeatles*复制到目录*binaries*，输入**make TARGET=install**。要清除该项目，输入**make TARGET=clean**。

讨论

这里所介绍的用于管理复杂项目的方法称为迭代式*make*。它允许你把一个项目组织成多个模块的集合，每个模块都有它自己的makefile文件，并可以指定模块之间的依赖关系。该方法不仅限于只有一个top-level makefile加一个子makefile集合；可以扩展为处理多

www.TopSage.com

层的树结构。尽管迭代式*make*曾经被认为是管理大型项目的标准方法，但现在已经有了更好的方法。有关的详细信息可以参见*Managing Projects with GNU make*, Third Edition (Robert Mecklenburg, O'Reilly)。

示例1-23是1.15节、1.16节和1.17节所介绍的技术的简单应用。这里真正令人感兴趣的只有一点。正如1.15节所示，当从命令行编译*hellobeatles.cpp*时，需要使用选项-I。这样，编译器就可以找到头文件*johnpaul.hpp*和*georgeringo.hpp*。一种解决方案是编写一个显式的规则，用含有选项*-I*的命令脚本来创建*hellobeatles.o*，如下所示：

```
hellobeatles.o: hello.beatles.cpp
    g++ -c -I.. -o hellobeatles.o hellobeatles.cpp
```

但是，这里使用的是用CPPFLAGS（如1.15节所示）来指定，当从*.cpp*文件编译目标文件时，就将把选项*-I*添加到如下命令行中：

```
CPPFLAGS      += -I..
```

这里使用的是赋值运算符+=而不是=，这样，只要在命令行或开发环境中指定了，CPPFLAGS的值就会进行累加。

现在让我们来看看示例1-24是如何工作的。其中最重要的规则是在*johnpaul*、*georgeringo*和*hellobeatles*目录中调用make：

```
johnpaul georgeringo hellobeatles:
    $(MAKE) --directory=$@ $(TARGET)
```

要理解这个规则，你需要了解这样三件事情。首先，变量MAKE扩展为当前运行的*make*。通常就是make，但在一些系统中称为gmake。第二，命令行选项*--directory=<path>*使用当前目录作为*<path>*的值来调用*make*。第三，一个含有多个目标的规则也就等于多个规则的集合，每个规则含有一个目标，并具有相同的命令脚本。因此，上面的规则就等价于：

```
johnpaul:
    $(MAKE) --directory=$@ $(TARGET)

georgeringo:
    $(MAKE) --directory=$@ $(TARGET)

hellobeatles:
    $(MAKE) --directory=$@ $(TARGET)
```

这又等价于：

```
johnpaul:
    $(MAKE) --directory=johnpaul $(TARGET)

georgeringo:
    $(MAKE) --directory=georgeringo $(TARGET)

hellobeatles:
    $(MAKE) --directory=hellobeatles $(TARGET)
```

因此，这个规则的结果是，在*johnpaul*、*georgeringo*和*hellobeatles*目录中，用变量

www.TopSage.com

TARGET的值加上命令行调用makefile文件。因此，你可以用选项*TARGET=xxx*运行top-level makefile文件，来创建每个subordinate makefile文件的目标*xxx*。

makefile最后的一个目标是确保subordinate makefile是按正确顺序运行的，它根据目标johnpaul和georgeringo声明目标hellobeatles：

```
hellobeatles: johnpaul georgeringo
```

在更复杂的应用程序中，可执行程序及其库之间可能会有很多依赖关系。在这种情况下，可以声明一个规则来表明它与其他库之间的依赖关系。

请参见1.5节、1.10节和1.13节。

1.19 定义宏

问题

如何定义预处理器符号名，并给它赋值为未定义值或已指定值？

解决方案

从命令行定义宏的编译器选项见表1-16所示。从IDE定义宏的指令见表1-17所示。要使用Boost.Build定义宏，只要给目标的需求项添加<define>*name*[=*value*]形式的属性，见前面的表1-15和示例1-12所示。

表1-16 从命令行定义宏

工具集	选项
所有工具集	*-Dname[=value]*

表1-17 从IDE定义宏

IDE	设置
Visual C++	从项目的属性页，切换到Configuration Properties ->C/C++-> Preprocessor，并在预处理器定义中输入*name[=value]*，多个项之间用分号分隔开
CodeWarrior	从Target Settings窗口，切换到Language Settings->C/C++ Preprocessor，并在标有Prefix Text的文本域中输入： **#define** *name[=value]*
C++Builder	从Project Options切换到Directories/Conditionals，并在预处理器定义中输入*name[=value]*，多个项之间用分号分隔开
Dev-C++	从Project Options中选择Parameters，并在C++ Compiler中输入： *-Dname[=value]*

www.TopSage.com

大学网 TopSage.com

讨论

在C++源代码中，经常使用预处理器符号，让源文件的单个集合可以与多个创建配置或操作系统使用。例如，假设你要编写一个函数，用来检查某个文件是否为目录。C++标准库没有提供完成这个工作的功能，因此你的函数需要使用平台的某些特性。如果你希望你的代码既可以在Windows系统下运行，也可以在Unix系统下运行，就需要确保当在Unix系统下编译代码时，与Windows有关的特性对编译器是不可见的，反之亦然。要实现这种要求的通常做法是使用条件编译，如示例1-25所示。

示例1-25 使用预定义宏进行条件编译

```
#ifdef _WIN32
# include <windows.h>
#else // Not Windows - assume we're on Unix
# include <sys/stat.h>
#endif

bool is_directory(const char* path)
{
#ifdef _WIN32
      // Windows implementation
#else
      // Unix implementation
#endif
}
```

在Windows系统下，除Cygwin之外的所有工具集都定义了宏_WIN32。以这种方式定义的宏就称为预定义宏。示例1-25使用预定义宏WIN32来确定是在哪种操作系统下编译，并启用相应平台的代码。

但是，执行某种条件编译所需的配置信息往往不能作为预定义宏。在这种情况下，就需要引入你自己的宏，使用表1-15、表1-16和表1-17所示的方法为它们赋给恰当的值。示例1-2是一个很好的例子。在Windows系统下，当创建DLL *georgeringo.dll*时，希望函数georgeringo()用属性__declspec(dllexport)进行声明，否则就用属性__declspec(dllimport)进行声明。如1.4节所示，当创建DLL时，你可以通过从命令行定义预处理器符号GEORGERINGO_DLL来实现，当编译使用该DLL的代码时则不必如此。

注意： 如果没有给宏指定一个值，大多数编译器会把值1赋给它，但其他编译器则赋给它空值。当用宏来启用条件编译时，如示例1-25所示，这种差别不重要。但是，如果你需要把宏扩展为特定的值，就应使用语法 *-D<name>=<value>* 来显式地赋值。

参见1.4节、1.9节、1.12节和1.17节。

www.TopSage.com

1.20 从 IDE 指定命令行选项

问题

如何给编译器或链接器传递一个命令行选项?

解决方案

很多IDE提供了一种给编译器或链接器直接传递命令行选项的方法。具体方法如表 1-18 和表 1-19 所示。

表 1-18 从 IDE 指定编译器选项

IDE	配置
Visual C++	从项目的属性页，切换到Configuration Properties→C/C++→-Command Line，并在 Additional options 下输入选项
CodeWarrior	无
C++Builder	无
Dev-C++	从 Project Options 选择 Parameters 并在 C++ Compiler 下输入选项

表 1-19 从 IDE 指定链接器选项

IDE	配置
Visual C++	从项目属性页切换到Configuration Properties→-Linker→-Command Line，并在 Additional options 下输入选项
Metrowerks	无
C++Builder	无
Dev-C++	从 Project Options 选择 Parameters 并在 Linker 下输入选项

讨论

尽管Visual C++通过图形界面提供了一些扩展配置选项，但仍允许显式地指定命令行选项。CodeWarrior 和 C++Builder 不允许显式地设置命令行选项，但这通常不是个问题，因为它们与 Visual C++ 一样，也是通过图形界面提供了一些扩展配置。但是，一些IDE除了提供在文本字段中显式地输入命令行选项之外，几乎没有配置命令行工具的其他方法。Dev-C++ 则介于这两者之间： Dev-C++ 为 GCC 工具集提供了比一些 IDE 更多的图形配置选项，但当使用 Dev-C++ 时，往往仍需要显式地输入命令行选项。

www.TopSage.com

1.21 生成调试版程序

问题

如何创建项目的调试版程序?

解决方案

通常，要生成一个调试版，必须：

- 禁用优化选项。
- 禁用内联函数扩展。
- 启用调试信息生成选项。

表1-20显示了通过编译器和链接器选项来禁用优化和内联函数。表1-21显示了通过编译器和链接器选项来启用调试信息。

表1-20 从命令行禁用优化和内联函数

工具集	优化	内联
GCC	`-O0`	`-fno-inline`
Visual C++ Intel (Windows)	`-Od`	`-Ob0`
Intel (Linux)	`-O0`	`-Ob0`
	`-opt off`	`-inline off`
Comeau (Unix)	`-O0`	`--no_inlining`
Comeau (Windows)	与后端相同，但是使用的是斜线（/）而不是横线（-）	
Borland	`-Od`	`-vi-`
Digital Mars	`-o+none -S`	`-C`

表1-21 启用调试信息的命令行选项

工具集	编译选项	链接选项
Comeau (Unix) *GCC Intel (Linux)* Metrowerks	*-g*	*-g*
Visual C++ Intel (Windows)	请参考表1-22	请参考表1-22

www.TopSage.com

表 1-21 启用调试信息的命令行选项（续）

工具集	编译选项	链接选项
Comeau (Windows)	与后端相同，但是使用的是斜线（/）而不是横线（-）	与后端的编译选项相同，但是使用的是斜线（/）而不是横线（-）
Borland	*-v*	*-v*
Digital Mars	*-g*	*-co*

表 1-22 启用 Visual C++ 或 Intel for Windows 的调试信息

编译选项	链接选项	IDE 选项	描述选项
-Z7	-debug	C7 兼容	调试信息存储在 .obj 和 .exe 文件中
-Zi [-Fd<pdb-file-for-obj>]	-debug[-pdb:<pdb-file-for-exe>]	程序数据库	调试信息存储在 .pdb 文件中；使用相同的选项来指定 .pdb 文件
-ZI [-Fd<pdb-file-for-obj>]	-debug [-pdb:<pdb-file-for-exe>]	Edit 和 Continue 的程序数据库	调试信息存储在 .pdb 文件中；使用相同的选项来指定 .pdb 文件。你的程序在一个调试会话中可以被重新编译

Boost.Build 为生成调试版提供了一种简单的方法：只需往目标需求中添加 `<variant>debug` 或使用命令行选项 *variant=debug* 即可。

一些 IDE 也提供了一种简单的方法来生成调试版。例如，当用 Visual C++ 创建一个新项目时，IDE 自动生成了调试版和发布版的配置信息。要请求调试版，只需从 Build 菜单中选择 Configuration Manager，并选择 Debug 作为当前配置。也可以从标准工具栏的下拉列表中选择 Debug。这样，当你下次创建项目时，就会生成一个调试版。

同样，当你在 CodeWarrior 中用 Metrowerks 的项目模板创建新项目时，IDE 也将自动生成调试版和发布版。调试版的名字可能会有所不同，但总是会含有 debug 一词。要创建调试版，从 Project 菜单中选择 Set Default Target，然后选择相应的菜单项。你也可以从项目窗口的目标下拉列表中选择调试版。

C++Builder 不支持单个项目的多个创建配置，但它为生成调试版提供了一个很容易的方法。要生成调试版，切换到 Project Option→Compiler，并按下 Full debug。这就可以禁用优化选项和内联选项并且启用调试信息。

如果你使用的 IDE 不提供预先设置的调试版和发布版配置，如 Dev-C++，或者你需要更多地控制你的项目设置，请参考表 1-23～表 1-25。

www.TopSage.com

表 1-23 给你的 IDE 禁用优化选项

IDE	配置
Visual C++	从项目的属性页，到配置属性→C/C++→优化并使得优化被禁用。并且设置这个页的其他属性值为默认设置
CodeWarrior	从目标设置窗口，到代码产生→全局优化并且设置它的值为禁用
C++Builder	从项目选项，到编译器并且在代码优化下面选择不优化
Dev-C++	从项目选项，到编译器→优化并且设置最少的优化次数为不优化；接着，到编译器→优化→进一步优化并且设置优化、更多的优化和最好的优化选项为不优化

表 1-24 给你的 IDE 禁用内联选项

IDE	配置
Visual C++	从项目属性页，到配置属性→C/C++→优化并且设置内联函数扩展为默认设置
CodeWarrior	从目标设置窗口，到语言设置→C/C++语言并且设置内联的深度设置为不扩展内联
C++Builder	从项目选项，到编译器并且禁用在调试下面的内联扩展
Dev-C++	请参考表 1-20 中的 GCC 描述和 1.20 节中的内容

表 1-25 给你的 IDE 启用调试信息

IDE	配置
Visual C++	请参考表 1-22
CodeWarrior	从目标设置窗口，到语言设置→链接器→PPC Mac OS X 链接器选择产生 SYM 文件和 SYM 文件中的完整路径选项
C++Builder	从项目选项，到编译器并且选择调试信息和行数信息选项
Dev-C++	请参考表 1-20 中的 GCC 描述和 1.20 节中的内容

讨论

所有的工具集都提供了一个选项来产生在一个对象文件和可执行文件中的信息，当一个程序一步一步运行时调试器能通过这些信息报告一些有用的信息。总的来说，这些信息包含源文件的文件名和对应于某个特殊对象或机器代码指令的代码行数，也包含那些关于 C++ 对象内存位置、对象名字和类型的信息。

大部分工具集都把调试信息存储在对象文件和可执行文件中，但也有一些工具提供一个选项来把产生的调试信息存储在一个分开的数据库文件中。例如，Visual C++ 中，-Z7 编

www.TopSage.com

大家网 TopSage.com

译选项指定这个调试信息应该存储在一个对象文件和可执行文件中，而编译选项 *-Zi* 则用来把调试信息存储在一个扩展名为 *.pdb* 的程序数据库文件中。编译选项 *-ZI* 则启动一个名为 *Edit* 和 *Continue* 的特性，它让IDE用户可以在不中断一个调试过程的情况下修改和编译他们的代码。同样，Mac 操作系统平台上的 CodeWarrior 默认的情况下就把调试信息存储到 *.SYM* 的文件中。

大部分的工具即使是在启动了优化和内联的选项时也能产生调试信息，尽管在某些情况下，这些调试信息可能与特定的优化不兼容。然而，当优化选项打开时，编译器还是有提高效率的自由的，它可以重新组织语句或者完全重新组织代码，只要看到的行为一致就可以。这就使得调试变得困难，因为在源代码的位置和对象或者机器代码之间可能不再那么地完全一致。对于内联来说，情况也相同：当编译器扩展一个内联函数时，与这个函数体相关的对象代码就在这个调用函数处产生，这就意味着调试器不能显示函数的参数值和局部变量。通常，调试器也不能报告对应于这个扩展处函数体的源代码位置。

因为这些考虑，通常当产生一个调试版的构建时是需要禁用优化和内联选项的。

请参考 1.22 节。

1.22 生成发布版程序

问题

为了发布给你的客户，如何生成一个小而快速的可执行文件或动态链接库？

解决方案

总的来说，为了产生一个发布版你必须：

- 打开优化选项。
- 打开内联函数扩展选项。
- 不产生调试信息。

表 1-26 说明了打开优化和内联的编译和链接选项。并没有关闭调试信息输出的命令行选项：当你从命令行构建程序时，默认情况下调试信息是关闭的。如果你使用 GCC 工具集，你可以通过给链接器指定 *-s* 选项来降低可执行文件和动态链接库的大小。

表 1-26 打开优化和内联的编译选项

工具集	优化	内联
GCC	`-O3`	`-finline-functions`
Visual C++ Intel	`-O2`	`-Ob1`

www.TopSage.com

表 1-26 打开优化和内联的编译选项（续）

工具集	优化	内联
Metrowerks	-opt full	-inline auto -inline level=8
Comeau (Unix)	-O3	
Comeau (Windows)	与后端相同，但是使用的是斜线（/）而不是横线（-）	--inlining
Borland	-O2	-vi
Digital Mars	-o+time	默认情况下是打开的

Boost.Build 为产生一个发布版提供了一个简单的机制：在你的目标需求中简单地增加<variant>release或者使用命令行选项*variant=release*，它能被简单地缩写为*release*。

一些IDE集成开发环境也给发布版构建提供了一个简单的方法。例如，就像我在1.21节中提到的那样，当你使用Visual C++来创建一个新的项目时，这个IDE集成开发环境将自动产生调试版和发布版的配置。如果你需要一个发布版的构建，你可以简单地从这个构建（build）菜单中选择配置管理器并且选择发布版（Release）作为活动配置。你也可以在一个标准的工具栏中从配置的drop-down列表中选择发布版（Release）。下一次你构建项目时，它将产生一个发布版的构建。

类似地，当你使用CodeWarrior并使用Metrowerks的项目模板来创建一个新的项目时，这个IDE集成开发环境将自动产生调试版和发布版构建。这个发布版目标的名字可能会有所不同，但它应该总是包括单词“release”或者“final”。为了得到一个发布版的构建，你可以从项目菜单中选择“Set Default Target”，然后选择相应的发布版目标。你也可以在你的项目窗口中的目标的“drop-down”列表中选择发布版。

C++Builder不支持在单个项目中提供多个构建的配置，但它给一个发布版的构建提供了一个简单的方法。为了配置一个发布版的构建，到“项目选项”→“编译器”并选择发布版。这将打开优化和内联选项并且不输出调试信息。

如果你使用的IDE集成开发环境没有提供预先设置好的调试版和发布版的配置，例如Dev-C++，或者你需要更多地控制你的项目设置，请参考表1-27～表1-29。

表 1-27 在IDE中打开优化选项

IDE	配置
Visual C++	从项目的属性页，到配置属性→C/C++→优化，并且设置Maximize Speed、Favor Size或Speed to Favor Fast Code和Global Optimizations、Enable Intrinsic Functions和Omit Frame Pointers选项都为yes。这页的其他属性设置都设为默认的值

www.TopSage.com

表 1-27 在IDE中打开优化选项（续）

IDE	配置
CodeWarrior	从目标设置窗口，到代码产生→全局优化并且选择4级选项
C++Builder	从项目选项，到编译器并且选择加速代码优化选项
Dev-C++	请参考表1-26中的GCC描述并参考1.20节

表 1-28 在IDE中打开内联选项

IDE	配置
Visual C++	从项目属性页，到配置属性→C/C++ →优化并且把内联函数扩展设置成“Any Suitable”
CodeWarrior	从目标设置窗口，到语言设置→C/C++语言。设置内联的深度为8，选择自动内联选项并且不选择其他的内联选项
C++Builder	从项目选项，到编译器并且不选禁用调试的内联扩展选项
Dev-C++	请参考表1-26中的GCC描述并参考1.20节

表 1-29 在IDE中关闭调试信息的输出

IDE	配置
Visual C++	从项目属性页，到配置信息→C/C++ > General并且选择禁用调试信息格式
Metrowerks	从目标设置窗口，到语言设置→链接器→x86链接器并且不选择存储完整路径、链接调试信息和调试内联函数
C++Builder	从项目选项，到编译器并且不选择“调试信息和行数信息”选项
Dev-C++	确保在命令行中 *-g* 选项没有指定，如1.20节描述的那样

讨论

大部分工具集都给优化提供几个选项；一些还提供了一打这样的选项。你所选择的优化极大地依赖于你项目的需求。例如，在一个嵌入式环境中，你可能会选择那些使得可执行文件更小但有损于运行速度的选项。而在另一些情况下，运行速度则是非常重要的。一些选项可能在这个平台上使得程序运行得更快但会使得程序在别的平台上运行得更慢。并且你也许会发现有些选项使得程序运行得更快而另一些则使得程序运行得更慢。

表1-26和表1-27中总的来说说明了那些一般意义上的优化选项，为了使程序更好地运行你还需要仔细考虑你的项目的需求，研究你的开发工具的文档并且做一些进一步的试验。

内联也是一样，尽管工具一般给内联提供的选项比优化更少。

请参考1.21节。

www.TopSage.com

1.23 指定运行库

问题

如何为你的工具集提供多个运行时链接库的支持，且为你的编译器和链接器指定某个使用的链接库？

解决方案

与一个给定的开发工具集一起提供的运行时链接库依赖于它们是单线程还是多线程，它们是静态库还是动态库，以及它们是否有调试信息和它们构建在一起。

如果你使用的是Boost.Build，表1-15中提供了三个选项来指定是否是多线程、运行时链接和可变的特性。例如，为了指定一个静态链接运行库，你可以在你的目标需求中增加`<runtime-link>static`，或者使用命令行选项*runtime-link=static*。为了指定多线程运行库，你就可以在你的目标需求中增加`<threading>multi`，或者使用命令行选项*threading=multi*。

如果你使用的是命令行方式的构建，你就可以使用表1-30～表1-36中的编译和链接选项。总的来说，调试版和发布版的链接库的命令行选项很相似；在这些表中，这些使用括号括起来的字母只是给调试版配置的。运行时链接库的动态变体的名字在圆括号中提供；如果你选择动态链接，这些链接库在运行时是可以使用的。

表 1-30 Visual C++或Intel (Windows)中运行时链接库的编译选项

	静态链接	动态链接
单线程	*-ML[d]*	n/a
多线程	*-MT[d]*	*-MD[d]* *(msvcrt[d].dll, msvcr80[d].dll)*

表 1-31 Metrowerks (Windows)中运行时链接库的编译选项

	静态链接	动态链接
单线程	*-runtime ss[d]*	n/a
多线程	*-runtime sm[d]*	*-runtime dm[d]* *(MSL_All-DLL90_x86[_D].dll)*

表 1-32 Max OS X平台上CodeWarrior运行时链接库的命令行选项

静态链接	动态链接
无	命令行选项请参考Metrowerks文档 *(MSL_All_Mach-O[_D].dylib)*

www.TopSage.com

表 1-33 Borland 运行时链接库的编译和链接选项

	静态链接	动态链接
单线程	*-WM*	*-WM- -WR -WC (cc3260.dll)*
多线程	*-WM*	*-WM -WR -WCa (cc3260mt.dll)*

表 1-34 Digital Mars 中运行时链接库的编译选项（所有的链接库都是多线程）

静态链接	动态链接
无	*-ND -D_STLP_USE_DYNAMIC_LIB (sccrt70.dll, stlp45dm.dll)*

表 1-35 GCC 中运行时库的链接选项

静态链接	动态链接
-static	无

例如，在 Visual C++ 的运行时链接库中使用编译选项 *-MD* 指定一个动态链接发布版。在 Windows 平台上指定一个静态链接的单线程的调试版的 Metrowerks 运行库，使用编译选项 *-runtime ssd*。为了指定一个单线程的动态链接的 Borland 运行时库，需要给编译器和链接器传递 *-WM- -WR -WC* 选项。

表 1-36 中描述了一些 IDE 中如何指定运行时链接库。

表 1-36 在你的 IDE 集成开发环境中指定运行时链接库

IDE	配置
Visual C++	从项目属性页，到配置属性 →C/C++→ 代码产生并且使用“drop-down”列表中的动态库
CodeWarrior	给动态库项目，增加对象文件 */usr/lib/dylib1.o* 和库 *MSL_Shared_AppAndDylib_Runtime[_D].lib* 和 *MSL_All_Mach-O[_D].dylib* 到你的项目中，并且删除任何 *MSL_<XXX>_Mach-O[_D].lib* 形式的库。 为可执行项目增加对象文件 */usr/lib/crt1.o* 和库 *MSL_Shared_AppAndDylib_Runtime[_D].lib* 和 *MSL_All_Mach-O[_D].dylib* 到你的项目中并且删除任何 *MSL_<XXX>_Mach-O[_D].lib* 形式的库
C++Builder	无论你的项目是单线程还是多线程，你都必须指明它。为了选择一个静态或者是动态运行时库，你可以到“项目选项”中的“链接器”选项并且选择使用或不使用动态 RTL

www.TopSage.com

大家网 TopSage.com

表 1-36 在你的 IDE 集成开发环境中指定运行时链接库（续）

IDE	配置
Dev-C++	就像 1.20 节中描述的那样，使用命令行选项 *-static* 可以指定一个静态链接运行库

讨论

一个运行时链接库一般包含那些别的程序运行时需要的实用函数的实现。一般来说，运行时链接库包含那些C标准库的函数实现，为访问操作系统服务如线程和文件系统的平台特有的函数，和C++语言特性的基础框架的函数如运行时类型信息（RTTI）和异常处理等。

在大部分情况下，你拥有的选择越多，就越好；然而，运行时链接库的选择性多也会带来一些问题。最主要的问题就是要确保一个程序的所有组成部分 —— 静态链接库、动态链接库和可执行文件 —— 都使用一个运行时链接库的某个单一的版本。如果不是，应用程序可能就会链接失败，或者可能就会在运行时出现一个很难诊断的问题。

注意： 当你使用第三方的链接库时，有时你可能不能控制这个链接库链接了某个运行时链接库的哪个版本。在这种情况下，你可能被迫在一个单一的应用程序中使用一个链接库的多个变体。

那么你应该如何来确定使用哪个运行时链接库呢？有两种选择 —— 单线程还是多线程，调试版还是发布版，这些都是很直接的。

如果你的项目是多线程，或者依赖于那些多线程的链接库的话，如果你的开发工具提供的话，你就必须使用这个运行时链接库的多线程版本。从一个多线程中调用一个运行时链接库的函数，如果这个运行时链接库不是多线程的话，你的程序可能会导致一个不可预知的运行时行为。同样地，如果你需要产生一个调试版的程序，在可能的情况下，你也应该使用运行时链接库的调试版。

最后一个选择就是使用静态链接库还是动态运行时链接库，这个选择还是比较困难的。使用一个静态链接运行时链接库有几个优点。首先，如果你不得不发布一个动态运行时链接库的话，由于只有你的应用程序用到的那些函数才被链接上，它能使得你的整个应用程序发布的量更小（然而，如果你知道在目标系统中这个动态的运行时库已经存在的话，链接一个静态时运行时库将可能使得你的发布量更大）。第二，通过链接一个静态运行时库，你能避免那些链接库版本问题，当在一个系统中一个动态库的多个版本存在时这个库的版本问题就可能出现。

然而，使用动态链接也是很有吸引力的。这是因为作为动态链接库的一个集合是一个很

www.TopSage.com

有效的组织应用程序的方法。首先，它使得应用程序的一部分能在不需要应用程序重新安装的前提下被更新。第二，Windows 平台上通过利用动态链接库的延迟装入的特性有时可以大大地提高应用程序的性能。但是由于一个应用程序的所有部件都应使用某个链接库的某个单一的版本，所以一旦一个应用使用某个单一的动态链接库，那么这个应用程序的所有部件都应该使用动态链接。这样结果就是使用动态链接运行时链接库使得应用程序更容易模块化。

大部分情况下，我建议你选择使用动态链接。然而，就像我前面说明的那样，有时静态链接可能更好一些。有时，不可能提前知道哪种类型的链接更合适，因为你不知道你写的链接库将来是如何被使用的。在这些情况下，一个共同的解决方案就是给你的链接库提供多个版本，依据运行时的不同而可以被链接。

请参考 1.4 节、1.5 节、1.21 节和 1.25 节。

1.24 严格遵循 C++ 标准

问题

如何使你的编译器仅能接受遵循 C++ 语言标准的程序？

解决方案

表 1-37 中列出了那些指定严格遵循 C++ 标准的命令行选项。而表 1-38 指出了使得你的 IDE 集成开发环境严格遵循 C++ 标准的指令。

注意： 表 1-6 中我引入的一些编译器的选项是一致的。前面有些例子程序已经包含打开语言基本特性的选项如宽字符支持、异常和运行时类型信息。在表 1-37 中，我就不再重复说明。

表 1-37 严格遵循 C++ 语言标准的一些命令行选项

开发工具	命令行编译选项
GCC	*-ansi -pedantic-errors*
Visual C++	*-Za*
Intel (Windows)	*-Za -Qms0*
Intel (Linux)	*-strict-ansi*
Metrowerks	*-ansi strict -iso_templates on -msext off*
Comeau (Windows)	*--A*
Comeau (Unix)	*--strict or -A*
Borland	*-A*
Digital Mars	*-A*

www.TopSage.com

表 1-38 使得你的IDE开发环境严格遵循C++语言标准的配置

IDE	配置
Visual C++	从项目属性页，到配置属性→C/C++→语言并且设置“Disable Language Extensions”、“Treat wchar_t as Built-in Type”和“Force Conformance in For Loop Scopes”选项都为yes
Metrowerks	从目标设置窗口，到语言设置→C/C++语言并且仅选择ISO Template Parser、ANSI Strict、ANSI关键字。并且确保这些编译选项能启用C++异常、启用RTTI支持、启用布尔型支持并且启用wchar_t支持
C++Builder	从“Project Options”，到“Advanced Compiler”并且选择“Language Compliance”下的“ANSI”
Dev-C++	请参考表1-37中GCC的选项和1.20节的内容

讨论

C++语言是在1998年由国际标准化组织（ISO）标准化的；并且在同一年，这个ISO标准也被美国国家标准协会（ANSI）所采纳。在2003年，这个标准的第二个修订版也被批准了；这个修订版只包含一些更正和澄清，并没有引入新的语言特性。目前正在做的工作是在这个C++标准的更新版本上进行的，它将包含一些新的语言特性和一个扩大的标准库。

在1998年这个标准被批准时，事实上没有编译器严格地遵循这个标准，尽管很多编译器都说自己是与ANSI标准一致的。然而，很多年过去了，这些编译器的提供商也做了大量的工作使得它们的工具符合这个标准。就像2005年9月那样，GNU、微软、intel、Metrowerks和Comeau编译器的最新版都是非常一致的。而Comeau和Intel以及和它们支持的输出模板现在基本上是100%一致的。

没有编译器能够非常好地遵循这个标准，也就是它拒绝任何不合法的程序。这不仅是因为没有编译器是100%一致的：更基本的原因是C++标准也不要求一个编译器拒绝所有的不合法程序。这就产生了一组仔细分界的情况，在这些情况下，编译器可能产生一个诊断信息来标明一个有点问题的程序；即使是很多不合标准的程序，也有可能不需要诊断信息。这就是程序调用了标准中称之为运行时未定义的行为。并且即使要求一个诊断信息，编译器也可能产生一个诊断信息后继续运行，并有可能成功地创建一个可执行文件或链接库。

编译器不要求一定抵制所有的非一致性程序的最主要的原因是，在很多情况下，非一致性的计算也是很困难的，或者是几乎不可能被发现的。稍后讨论的另一个原因是非一致性程序也是很有用的。

www.TopSage.com

只要有可能，我还是建议你使用编译器严格遵循标准的选项。然而还是存在某些情况下这样也是不合适的，为了更好地理解这一点，让我们来看看几个不同类型的非一致性代码。

首先，在C++被标准化之前，还有很多的代码使用很多C++的方言实现的，现在这些代码还在使用着。例如，在C++的早期，在一个for循环中的初始化中声明的变量的作用域是可以扩展到包含这个循环的代码块尾部：

```
// WARNING: invalid code!
int main(  )
{
    for (int i = 0; i < 10; ++i)
    ;
    int j = i; // j == 10
}
```

在标准中，这样的代码是不允许的，并且比起标准的作用域规则，这样也没有任何的优点。这样的代码需要被编译仅仅只能在维护过去留下来的代码时才可以。

另外一种非一致性的代码就是使用了那些试验性的语言扩展的代码，这些试验性的扩展最终可能并不加入到C++的标准中。例如，很多编译器都提供了一种整型类型long long来支持最少64位的宽度。另一个例子就是，有几个编译器提供了一个内置操作符typeof，它和sizeof操作符具有相同的语法，都返回一个表达式的类型。在C++标准的下一个版本中，很有可能两者都会出现，尽管这个typeof的写法希望被修改，有可能变成decltype。

但是在使用这样的语言扩展性之前还是小心：在你知道之前，你可能需要把你的代码移植到不支持这个扩展性的平台上，或者用不同的语义实现它们的平台上。

第三类非一致性代码就是为了利用操作系统的特性使用了特定于某个平台的语言扩展。属性__declspec(dllexport)和__declspec(dllimport)就是为了在windows平台上建立动态链接库，而属性__stdcall、__fastcall和__cdecl则代表Windows上调用惯例的一类非一致性。尽管这些是语言的扩展，但在windows平台上的大部分编译器即使是在最严格的标准遵循的模式下也是接受的。

非一致性代码的最后一类就是代码违反C++标准但根据一些别的有用的标准则是非常合法的。这样的一个很主要的例子就是C++/CLI标准，这是目前由ECMA标准化的最后一个阶段。C++/CLI是C++的一个扩展，它组成了命令行语言框架的C++接口，这也是微软的.NET框架的核心。当编译一个这样的使用了某种C++/CLI扩展的应用程序时，一个遵循C++标准的编译器就会报告一个诊断信息，但是如果这个编译器支持C++/CLI标准的话，它就能很容易地产生合法的C++/CLI程序。

如果你需要编译非一致性代码的话，首先你可以在表1-37和表1-38中查找，看能否使用这些编译选项来编译它。如果不能的话，别的一些提供了更多的支持稍微不遵循标准的

www.TopSage.com

编译选项的编译器，你可以试试。例如，Comeau提供了*long_long*选项来指明支持long long类型。最后，也有一些编译器提供一些选项来作为警告而不是错误报告那些违反标准的代码。例如，GCC就提供了选项pedantic，Comeau就为Windows平台提供了选项a，给别的平台提供了*strict_warnings*或*a*选项。

请参考1.2节。

1.25 使源文件自动链接到指定的库

问题

你已经实现了一个库，想用一个头文件的集合和已编译的静态或动态链接库来发布它，如何使你的库用户在链接他们的应用时无需指定库的名字？

解决方案

如果你在Windows平台上编程并且使用Visual C++、Intel、Metrowerks、Borland或者Digital Mars工具集的话，你就可以在你的库的头文件中使用pragma注释来指定库的名字，并且你也可以不指定完整路径的文件名，那些包含这个头文件的任何代码都可以自动链接这个库。

例如，假设你想以一个静态链接库*libjohnpaul.lib*和头文件*johnpaul.hpp*一起来发布示例1-1中的库。你就可以如示例1-26中这样修改这个头文件：

示例1-26 使用pragma注释

```
#ifndef JOHNPAUL_HPP_INCLUDED
#define JOHNPAUL_HPP_INCLUDED

#pragma comment(lib, "libjohnpaul")

void johnpaul( );

#endif // JOHNPAUL_HPP_INCLUDED
```

这样做完修改后，当链接代码包含这个头文件*johnpaul.hpp*时，Visual C++、Intel、Metrowerks、Borland和Digital Mars的链接器就能自动找到这个库*libjohnpaul.lib*。

讨论

在某些方面，链接比起编译来说是构建过程中更为困难的阶段。链接过程中一个最为共同的问题是当链接器发现了库的一个错误版本时该如何处理呢。在Windows平台上，这尤其是一个特殊的问题，运行时库和那些依赖这些库的库经常都有很多变体。由于这个原因，Windows上的库都通过名字上的标识来发布从而反映各种不同的构造配置。虽然这有助于减少版本之间的冲突，但是它也使得链接更为困难，因为你不得不给链接器指定正确的名字标识。

www.TopSage.com

大家网 TopSage.com

由于这个原因，pragma 注释是一个强大的工具。在这些事情中，它让你在一个头文件中指定一个库的正确的名字标识，使得用户不用去明白你的名字标识上的惯例。此外，如果你设计你的安装过程来把这些二进制文件复制到由链接器自动找到的位置，如 Visual C++、CodeWarrior 的 *lib* 子目录，或者是 C++Builder 的根目录，程序员只需要简单地包含你的头文件就能使用你的库。

到目前为止，还是很好的。仅有一个问题：pragma 注释并不是所有编译器都支持的。如果你需要实现一个可移植的代码，你就只能是确定你的开发工具支持 pragma 注释时才调用 pragma。例如，你可以如下这样修改 *johnpaul.cpp*：

```
#ifndef JOHNPAUL_HPP_INCLUDED
#define JOHNPAUL_HPP_INCLUDED

#if defined(_MSC_VER) || \
    defined(_  _ICL) || \
    defined(_  _MWERKS_  _) && defined(_WIN32) || \
    defined(_  _BORLANDC_  _) \
    defined(_  _DMC_  _) \
    /**/
# pragma comment(lib, "libjohnpaul")
#endif

void johnpaul();

#endif // JOHNPAUL_HPP_INCLUDED
```

这个例子已经非常复杂了，但是不幸的是，它还是不够精确：一些不支持 pragma 注释的编译器为了与 Visual C++ 相兼容定义了宏 _MSC_VER。幸运的是，Boost 提供了一个简单的解决方案：

```
#ifndef JOHNPAUL_HPP_INCLUDED
#define JOHNPAUL_HPP_INCLUDED

#define BOOST_LIB_NAME libjohnpaul
#define BOOST_AUTO_LINK_NOMANGLE
#include <boost/config/auto_link.hpp>

void johnpaul(  );

#endif // JOHNPAUL_HPP_INCLUDED
```

下面这行代码：

```
#define BOOST_LIB_NAME libjohnpaul
```

指明了你的库的名字，而下面这行代码：

```
#define BOOST_AUTO_LINK_NOMANGLE
```

这就表明你不想使用 Boost 的名字标识惯例，而下面这行代码：

```
#include <boost/config/auto_link.hpp>
```

则为那些支持 pragma 注释的编译器调用 pragma 注释。

请参考 1.23 节。

www.TopSage.com

1.26 使用输出模板

问题

如何构建一个使用了输出模板的程序，也就是说，在头文件中使用export关键字来声明模板并且把模板的实现放在*.cpp*文件中？

解决方案

首先，把这个包含模板实现的*.cpp*文件编译成对象文件，为了输出模板你需要给编译器传入那些必须的命令行选项。接下来，编译并链接这个使用了输出模板的*.cpp*文件，当然你还需要给编译器和链接器传入一些为了输出模板所需的命令行选项和指定包含这个模板实现的目录的选项。

表1-39中给出了一些输出模板的选项。表1-40则指出了那些指定模板实现位置的选项。如果你使用的工具集没有出现在这两个表中的话，很有可能它不支持输出模板。

表1-39 输出模板的一些编译选项

工具集	脚本
Comeau (Unix)	*--export*, *-A* 或 *--strict*
Comeau (Windows)	*--export* 或 *--A*
Intel (Linux)	*-export* 或 *-strict-ansi*

表1-40 指定模板实现位置的编译选项

工具集	脚本
Comeau	*--template_directory=<path>*
Intel (Linux)	*-export_dir<path>*

例如，假设你想编译示例1-27中的程序。它由三个文件组成：

- 文件*plus.hpp*包含这个输出函数模板`plus()`的声明。
- 文件*plus.cpp*包含*plus()*的定义。
- 文件*test.cpp*包含`plus()`的声明而不是它的定义，并且定义了一个使用`plus()`的`main()`函数。

示例1-27 一个简单的使用输出模板的程序

plus.hpp:

```
#ifndef PLUS_HPP_INCLUDED
#define PLUS_HPP_INCLUDED
```

www.TopSage.com

```
export template<typename T>
T plus(const T& lhs, const T& rhs);

#endif // #ifndef PLUS_HPP_INCLUDED
```

plus.cpp:

```
#include "plus.hpp"

template<typename T>
T plus(const T& lhs, const T& rhs)
{
    return rhs + lhs;
}
```

test.cpp:

```
#include <iostream>
#include "plus.hpp"

int main( )
{
    std::cout << "2 + 2 = " << plus(2, 2) << "\n";
}
```

在unix平台上使用Comeau来把这个*plus.cpp*编译成对象文件*plus.obj*，并且改变到包含*plus.cpp*、*plus.hpp*和*test.cpp*的目录，并输入如下这样的命令：

```
$ como -c --export plus.cpp
```

这个命令同时还产生一个*plus.et*文件，它描述了这个包含在*plus.cpp*文件中的模板实现。

注意：如果你有兴趣的话，可以用一个文本编辑器打开它。

接下来，把这个*test.cpp*文件编译成一个对象文件*test.obj*，你可以如下这样做：

```
$ como -c --export test.cpp
```

最后，链接成一个可执行文件*test.exe*：

```
$ como --export -o test test.obj
```

这后面两个命令可以合并到一起，如下所示：

```
$ como --export -o test test.cpp
```

你可以运行这个程序：

```
$ ./test
  2 + 2 = 4
```

你也可以假设文件*plus.hpp*和*plus.cpp*在一个名为*plus*的目录中，而*test.cpp*在同级的名为*test*的目录中。为了编译和链接*test.cpp*文件，你可以把当前目录改到*test*目录并输入如下命令：

```
$ como --export --template_directory=../plus -I../plus -o test
test.cpp
```

www.TopSage.com

讨论

C++提供了两个模型来支持函数模板和类模板的静态数据成员的定义：包含模型和独立模型。对那些经常使用C++模板的编程人员来说包含模型是很熟悉的，但对于那些不经常使用模板编程的开发人员来说则不是那么自然。在这个包含模型中，一个函数模板的定义或者是一个类模板的静态数据成员则必须被每一个使用它的源文件所包含。相反，对于非模板函数和数据成员来说，源文件只用简单地包含它的声明就可以；它的定义可以放在一个独立的*.cpp*文件中。

独立的模式则与传统的C++组织源文件的方式更相近。使用`export`关键字声明的模板的定义不需要被使用它们的源文件所包含；相反，这个定义可以放在一个独立的*.cpp*文件中。和传统的源文件组织并行不够精确，因为即使这些使用一个导出模板的代码不需要包含它的定义，但是它在某些微妙的地方仍然依赖这个定义。

这个独立模式具有几个潜在的优点：

减少了编译时间

独立模式可以减少编译时间是因为模板的定义不会那么经常地被检查，并且独立模式可以减小模块之间的依赖。

减少了符合污染

在一个模板实现文件中使用的数据、类和函数的名字可以被隐藏起来，那些使用这个模板的代码是看不到的，这样就减少了偶尔的名字冲突。此外，模板实现的作者不用担心这些模板实现中的名字和使用这个模板的源文件中名字的偶尔冲突。

打包一个预编译模板的实现的能力

理论上，在一个独立的模式中，一个厂商可以打包一个介于C++源文件和通常的对象文件之间的预编译成的二进制格式的模板实现。

独立模式的这三个潜在的优点都是有争议的。首先，在一些使用报告减少编译时间的同时，独立模式在某些情况下也可能导致更长的编译时间。到目前为止，还没有足够的证据来支持这个判断。第二，独立模式在减少符合污染的同时，语言的规则也必须支持这个独立模式，特别是这个两阶段查找的概念，使得实现模板的方式更复杂，即使是使用包含模式，也有可能产生一些不可预知的后果。第三，所有这些独立模式的现有实现都基于EDG前端，而EDG还没有提供任何的方式来把包含输出模板实现的源文件编译成二进制并且这些二进制文件可以在源文件的场合打包的方法。

2003年，一度有人想从未来的C++标准版本中把输出模板去掉，但是没能成功。所以，输出模板将是C++语言中永恒的一部分，因此你还是应该学会使用它们。

参见1.25节。

www.TopSage.com

第2章

代码结构

2.0 概述

C++如此受欢迎的原因之一或许是它可以开发小型、大型和巨型项目的能力。你可以为一个小型的原型或研究项目编写一些类，随着项目的增大和开发人员的增多，C++允许你把该应用程序扩展成多个模块，这些模块之间各自独立。这样做的代价是你需要花时间进行手工的结构重组（如添加命名空间、重新安排头文件的物理位置等）。但通常这是值得的，因为你可以使程序模块化，使不同的开发人员只关注他们自己要进行的编码和功能。

你需要投资的这种手工劳动与你设计模块所花费的时间是成反比例的。一开始就使用一些良好的模块化技巧，你的代码就有更好的可扩展性。

如果你还没有使用过命名空间，那么你至少已经听说过它们，你很有可能已经使用过一个：std命名空间。这是一个包含有标准库的命名空间。以我的经验，命名空间的使用并不很频繁，这不是因为它们很复杂，或使用它们需要很多的努力。2.3节将介绍如何用命名空间来模块化代码。

本章中的很多技巧描述了在头文件中使用的技术。由于要讨论不少内容，而且每个内容介绍头文件的不同部分，因此，在示例2-1中将介绍一个典型的头文件，它使用了本章要介绍的所有技术。

示例2-1 一个头文件

```
#ifndef MYCLASS_H_ _ // #include guards, Recipe 2.0
#define MYCLASS_H_ _

#include <string>
namespace mylib { // Namespaces, Recipe 2.3

   class AnotherClass; // forward class declarations, Recipe 2.2
   class Logger;
```


www.TopSage.com

大家网 TopSage.com

```
   extern Logger* gpLogger; // External storage declaration, Recipe 2.1

   class MyClass {
   public:
 std::string getVal() const;
 // ...
   private:
 static int refCount_;
 std::string val_;
   }

// Inline definitions, Recipe 2.4
inline std::string MyClass::getVal() const {
   return(val_);
}

#include "myclass.inl"

} // namespace

#endif // MYCLASS_H_ _
```

一旦你编写的头文件不是这种样子，大多数时候你就需要一个含有定义而不只是声明的实现*.cpp*文件。通常是把定义放在头文件而不是实现文件中，但为了保持完整性，示例2-2给出了一个实现文件样本，它使用了示例2-1的头文件。

示例2-2 一个实现文件

```
#include "myclass.h"

namespace mylib {

   MyClass::refCount_ = 0; // Static definition, Recipe 8.4

   MyClass::foo() { // Method implementations
 // ...
   }
}
```

当然，你的实现文件不会包含详细的注释，这里是为了清楚起见。

2.1 确保头文件只包含了一次

问题

你有一个头文件，它需要被几个其他的文件包含。如何在这个头文件中使用预处理器，使得这些声明只被扫描一次？

解决方案

在你的头文件中使用#define来定义一个宏，并且只在这个宏没有被定义时包含这个头文件中的内容。你可以使用#ifndef、#define和#endif预处理指令的集合，如示例2-1中所做的那样：

```
#ifndef MYCLASS_H_ _ // #include guards
#define MYCLASS_H_ _
```

www.TopSage.com

```
// Put everything here...
#endif // MYCLASS_H_ _
```

当预处理器检测这个头文件时，这个预处理器首先将碰到的内容之一就是这个#ifndef预处理指令和跟在它后面的符号。只有在这个符号MYCLASS_H__还没有被定义时，#ifndef才告诉预处理器继续处理接下来的行。如果这个宏已经被定义，那么预处理器就应该跳过后面的代码直到碰到#endif预处理指令。紧接在#ifndef预处理指令后面定义了MYCLASS_H__宏，因此如果在同一次编译期间这个文件被预编译器检查了两次的话，在第二次时这个宏MYCLASS_H__就已经被定义了。把你的代码放在#ifndef和#endif之间，就能保证在编译时这些代码只被读一次。

讨论

如果你不使用这项称为包含保护的技术，你就可能已经碰到了“symbol already defined”的编译错误，这个错误给多个重复的定义提供了保护性的措施。C++不允许你定义相同的符号两次或两次以上，并且如果（为某种目的或者是偶然这样）这样做的话，你都将得到一个编译错误。包含保护阻止了这种错误，并且它们是一种非常标准的技术实践。

这个你使用#define定义的宏不必遵循某种特殊的格式，但是上面我使用的语法是普遍的。这个目的就是使用一个不会与别的宏相矛盾的符号并且使得你的文件在预编译时可以不被注意地跳过。在实践中，你或许注意到了另一种技术，例如在一个宏中包含一个头文件或模块的版本信息，例如，MYCLASS_H_V301_ _或者作者的名字等。对于你如何来命名宏都是不重要的，只要它们被一致地使用就可以。这些宏只应该在这些需要保护的文件而不是别的地方被引用。

在一些代码中，或许你看到了外部的包含保护，这个和前面描述的内部包含保护是相同的，只不过它们出现在这些包含这个头文件的文件中而不是出现在头文件本身中：

```
#ifndef MYCLASS_H_ _
#include "myclass.h"
#endif
```

如果这个宏MYCLASS_H_ _已经被定义的话，myclassh.h头文件就不会被包含进去。外部的包含保护在过去的很多年中被认为是在大型项目中减少编译时间的方法，但是由于编译器得到了改进，现在这项技术已经是不必要的。请不要使用它。

即使你正工作在一个小型项目中，在头文件中使用包含保护也是一个好的主意。如果你的头文件被别的多个文件所包含，你总有一天会得到重复定义的错误。更为重要的是，在一段时间后，这些小项目都很有可能变成大项目，而一个开始只有单个可执行文件和一序列相互包含一次的头文件的项目迟早都会变大并且产生编译错误。如果你在一开始就增加使用包含保护，你就不用在将来的某个时刻回到这些文件中来给这一系列的文件增加包含保护。

www.TopSage.com

2.2 保证一个跨多个源文件的变量只有一个实例存在

问题

如何使得在同一程序的多个不同模块中使用同一个变量，并且只能有这个变量的一个副本？换句话说，也就是一个全局变量。

解决方案

在一个单一的实现文件中以一种通常的方式来声明和定义这个变量，并且在其他的实现文件中当你需要在运行时访问这个变量时使用extern关键字。通常，这就意味着在一个头文件中包含这个extern声明，而这个头文件则需要被所有那些需要访问这个全局变量的实现文件所使用。示例2-3包含了一些文件，这些文件说明了这个extern关键字是如何被用来访问定义在别的实现文件中定义的变量的。

示例2-3 使用extern关键字

```
// global.h
#ifndef GLOBAL_H_ _ // See Recipe 2.0
#define GLOBAL_H_ _

#include <string>

extern int x;
extern std::string s;

#endif

// global.cpp
#include <string>

int x = 7;
std::string s = "Kangaroo";

// main.cpp
#include <iostream>
#include "global.h"

using namespace std;

int main() {

   cout << "x = " << x << endl;
   cout << "s = " << s << endl;
}
```

讨论

这个extern关键字是一种告诉编译器这个变量的事实上的存储空间在别的地方已经分配的方法。extern关键字也告诉链接器它查验的这个变量存在于别的对象文件中，并且当链接器创建最后的可执行文件或库时，它需要到别的文件中找到这个变量。如果链接器永远都不能找到你声明的这个外部变量，或者它找到了这个变量的两个或两个以上的定义，那么就会报一个链接错误。

www.TopSage.com

示例2-3并不惊奇，但它非常好地说明了这点。我的两个全局变量都声明和定义在*global.cpp*文件中：

```
int x = 7;
std::string s = "Kangaroo";
```

我需要从别的其他的实现文件中访问它们，因此我在头文件*global.h*中用一个extern关键字来声明它们：

```
extern int x;
extern std::string s;
```

声明和定义之间的区别是非常重要的。在C++中，你可以多次声明一个变量，只要这个声明能够匹配得上，但是你只能定义一次；这就是一次定义规则（在某些情况下，事实上你也可以多次定义，但是这些定义都是要严格相同的，通常这不是一个好的主意）。这个extern关键字就是一个机制，通过这个机制来告诉编译器和链接器它的定义存在于别的地方并且需要在链接时解析。

这不是说你就应该经常地使用extern关键字。事实上你还是应有所保留地使用它，并且只能是你不得不这样时才使用，因为它允许全局于应用的变量。有时你或许真的需要一个这样的全局对象或数据，如一个日志对象、一些硬件、一个大的共享数据对象，但是大部分情况下，这还是有更适合的设计可以替代的。

2.3 使用向前的类声明来减少 #include 的使用

问题

你有一个头文件，它引用了别的头文件中的类，并且你又需要减少编译依赖（和也许还有时间）。

解决方案

在可能需要避免不必要的编译依赖的地方使用向前类声明。示例2-4给出了一个向前类声明的简单例子。

示例2-4 向前的类声明

```
// myheader.h
#ifndef MYHEADER_H_ _
#define MYHEADER_H_ _

class A; // No need to include A's header

class B {
   public:
 void f(const A& a);
   // ...
   private:
```

www.TopSage.com

```
 A* a_;
};

    #endif
```

在别的地方也许存在一个头文件和一个实现文件来声明和定义类A，但从这个*myheader.h*内部来说，我不用关心这个类A的细节：我所需要知道的就是这个A是一个类而已。

讨论

一个向前的类声明是一个忽略细节的方法，而这些细节又是你不需要关心的。在示例2-4中，*myheader.h*除了需要知道类A确实是一个存在的类之外不需要别的任何信息。

如果你使用#include来包含类A的头文件的话，你觉得会发生什么，或者更现实一点，在一个现实的头文件中，你可能需要使用六、七个类的头文件。这样，一个实现文件（*myheader.cpp*）将会包含这个头文件，*myheader.h*，因为这个头文件包含所有的声明。到目前为止还不错。但是如果你改变这个*myheader.h*包含的任何一个头文件（或者你改变这些文件中的某一个所包含的头文件）的话，那么所有这些包含这个头文件的实现文件将不得不重新编译。

向前声明你的类的话，那么这些编译依赖将不复存在。使用向前声明简单地创建名字，任何在这个头文件中的类都可以引用这个名字。链接器具有根据这个你在头文件中使用的名字来匹配实现文件中的定义的功能。

不幸地是，你并不是可以一直使用向前声明。示例2-4中的类B仅使用了类A的指针或引用，因此你可以使用一个向前声明就可以。然而，如果我使用了类A的成员函数或成员变量的话，或者如果在这个类B的定义中我有一个类型A的对象而不是类A的一个指针或引用的话，我的这个向前声明就不够了。这是因为这些包含*myheader.h*头文件的文件需要知道B的大小，并且如果A是B的一个成员变量的话，编译器需要知道A的大小来计算出B的大小。一个指针或引用总是具有固定的大小，因此在那些你使用指针或者引用的场合，编译器不需要关注A的细节，因此A的头文件就是不必要的。

不用惊奇，如果你在*myheader.h*头文件中包含任何使用了类A的成员的定义，你将不得不包含类的头文件。这是因为编译器需要检查你引用的这个类A的成员函数的函数签名或者这个类A的数据成员的数据类型。作为一个解释，下面的代码就需要一个#include：

```
#include "a.h"

class B {
   public:
 void f(const A& a) {
    foo_ = a.getVal(  ); // Have to know if a.getVal is valid
 }
// ...
```

总的来说，当你需要减少这些在编译时运行的#include时，请使用向前声明。

www.TopSage.com

大家网 TopSage.com

2.4 使用命名空间来阻止名字冲突

问题

你在不同的模块中使用的名字可能会产生冲突，或者你想提前创建一些代码逻辑组来避免这样的冲突出现。

解决方案

使用命名空间来模块化代码。使用命名空间，你就能把一些大的分布在多个独立的文件中的代码组成到一个单一的命名空间中。并且你也可以依据需要来嵌套使用命名空间来把一个大的模块分成很多子模块，并且你的模块的用户就可以有选择地使用他们需要使用的在你的命名空间中的元素。示例 2-5 描述了一些你在一个命名空间中可以使用的方法。

示例 2-5 使用命名空间

```
// Devices.h
#ifndef DEVICES_H_ _
#define DEVICES_H_ _

#include <string>
#include <list>

namespace hardware {

   class Device {
   public:
 Device() : uptime_(0), status_("unknown") {}
 unsigned long getUptime() const;
 std::string getStatus() const;
 void reset();
   private:
 unsigned long uptime_;
 std::string status_;
   };

   class DeviceMgr {
   public:
 void getDeviceIds(std::list<std::string>& ids) const;
 Device getDevice(const std::string& id) const;
 // Other stuff...
   };
}

#endif // DEVICES_H_  _

// Devices.cpp
#include "Devices.h"
#include <string>
#include <list>

namespace hardware {

   using std::string;
   using std::list;

   unsigned long Device::getUptime() const {
 return(uptime_);
```

www.TopSage.com

```
    }
    string Device::getStatus() const {
 return(status_);
    }

    void DeviceMgr::getDeviceIds(list<string>& ids) const {
    }

    Device DeviceMgr::getDevice(const string& id) const {
 Device d;
 return(d);
    }
}

// DeviceWidget.h
#ifndef DEVICEWIDGET_H_ _
#define DEVICEWIDGET_H_  _

#include "Devices.h"

namespace ui {

    class Widget { /* ... */ };
    class DeviceWidget : public Widget {
    public:
 DeviceWidget(const hardware::Device& dev) : device_(dev) {}
 // Some other stuff
    protected:
 hardware::Device device_;
    };
}
#endif // DEVICEWIDGET_H_  _

// main.cpp
#include <iostream>
#include "DeviceWidget.h"
#include "Devices.h"

int main() {

    hardware::Device d;
    ui::DeviceWidget myWidget(d);
    // ...
}
```

讨论

示例2-5有点复杂，让我们逐步来解释它，因为它说明了关于命名空间的几个关键点。假设你正在写一个管理程序，它需要与硬件设备的接口打交道。你想把你的应用程序分成两个或更多个命名空间以阻止名字上的冲突，或者仅是从逻辑上把这个应用程序分成两个部分，通过这种方法就能使人更好地理解。

首先，考虑文件*Devices.h*，它包含几个用来管理硬件元素的类：Device和DeviceMgr。我不想把它们放在全局命名空间中（这就意味着它们的名字在程序的任何地方都可见），因此我把它们放到一个名为hardware的命名空间中：

```
#ifndef DEVICES_H_ _ // See Recipe 2.0
#define DEVICES_H_ _

#include <string>
#include <list>
```

www.TopSage.com

```
namespace hardware {

   class Device {
// ...
   };

   class DeviceMgr {
// ...
   };
}

#endif // DEVICES_H_ _
```

这个机制是简单的：在一个命名空间namespace块中包含你所想放到这个命名空间中任何东西。

上面的摘要是关于Device和DeviceMgr的声明的，但是我还是需要考虑一下它的实现，也就是出现在*Devices.cpp*中的内容。同样这样，在一个命名空间代码块中包含任何东西，把任何你需要的加到这个hardware命名空间代码块中。

```
#include "Devices.h"
#include <string>
#include <list>

namespace hardware {

   using std::string;
   using std::list;

   // Implementation for Device and DeviceMgr
}
```

这时候，这个hardware命名空间包含任何我们需要的东西。接下来需要做的就是在别的地方使用它。当然有很多方法可以这样做；示例2-5中我用命名空间的名字和这个Device类的名字一起来引用它，如下所示：

```
#ifndef DEVICEWIDGET_H_ _
#define DEVICEWIDGET_H_ _

#include "Devices.h"

namespace ui {

   class Widget { /* ... */ };
   class DeviceWidget : public Widget {
   public:
DeviceWidget(const hardware::Device& dev) : device_(dev) {}
// Other stuff...
   protected:
hardware::Device device_;
   };
}
#endif // DEVICEWIDGET_H_ _
```

在*main.cpp*的main函数中我也同样这样做：

```
int main() {
   hardware::Device d;
   ui::DeviceWidget myWidget(d);
}
```

www.TopSage.com

为了给这两个命名空间中任何一个增加类型，你可以如示例2-5那样使用同样的方法来声明你的头文件和实现文件。每次在一个命名空间块中包含你的代码，这就把这些代码加到这个命名空间中，因此你就可以拥有一些在这同一个命名空间中的代码，并且这些代码不用知道这个命名空间中任何别的东西。

你如果使用命名空间使类名合格，你很快就会觉得这很繁琐。当然还是有很多方法可以避免这样做的。你可以给这个命名空间创建一个别名，使用如下的关键字来这样做：

```
using hardware::Device;

int main() {
   Device d;  // No namespace name needed
   ui::DeviceWidget myWidget(d);
}
```

在后续的代码中，你可以简单地引用这个别名来代替完整的命名空间名称。或者你可以通过使用这个命名空间而不是使用它包含的每一个元素来引入这个命名空间中的任何东西：

```
using namespace hardware;

int main() {
   Device d;
   ui::DeviceWidget myWidget(d);
}
```

你或许已经使用过这项技术，或者当你使用标准类库（本书中的很多例子都使用了这项技术）时，至少在一些例子中看到过。任何在标准类库中的东西都是处于std命名空间中，因此你将经常看到如下这行代码：

```
using namespace std;
```

引入整个命名空间常常是一个不好的主意，并且总的来说这样被认为是一个不好的实践。本书中的大部分例子为了清晰性都引用了这个std命名空间，但是总的来说，在现实的程序中，还是不推荐这样做。

如果你引入整个命名空间或者它们中的某几个，那么命名空间的效用就会大大地降低。命名空间存在的原因之一就是减少命名上的冲突。如果你引入了很多不同的命名空间，那么你就增加了命名冲突的可能性。你的代码可以编译和运行得很好，但是将来的某个时候，某个人可能给这个命名空间增加东西从而下一次编译时命名上的冲突就会出现。

你也可以以嵌套的方式来使用命名空间从而把一个命名空间分成很多小的组。例如，示例2-5中的我定义的hardware命名空间事实上就包含很多网络类和很多设备类，因此我就可以嵌套地使用命名空间来把这个hardware命名空间分成更多小的命名空间：

```
namespace hardware {
   namespace net {
 // network stuff
   }
```

www.TopSage.com

```
    namespace devices {
  // device stuff
    }
}
```

现在，我就可以使用更多的限定词来访问这个命名空间中的元素：

```
// In some other file...
using hardware::devices::Device;
```

命名空间是便捷的。你可以用命名空间来处理很多很好的事情从而使得你工作的很惬意：命名空间别名、在函数参数命名空间中自动的名字查找，以及函数重载当中名字的匹配等。后两项是冗长的，但也是简单的。

一个命名空间别名看上去是这样的：一个较为简短的名字，你通过这个较为简短的名字来代替另一个较为冗长的命名空间名字。如果你不想使用using语句，而且你每次使用某个类时你还不想输入这个类的全名，这时你就可以给这个命名空间创建一个别名：

```
using dev = hardware::devices;
// ...
dev::Device d;
```

当你需要引用这个命名空间中的元素时你就可以使用这个别名。

C++也在函数参数命名空间中提供了自动查找功能。因此，例如，下面的代码就是用一个命名空间来验证它的参数（Device类是声明在这个dev命名空间中）：

```
void f(dev::Device& d) {
   register(d);  // This is actually dev::register
}
```

当你传入一个属于某个命名空间的函数参数时，在这个函数体的内部执行这个函数的名字查找功能时编译器就会包含这个命名空间。你可能并不是每天都需要这样做，但是当你这样做时，它确实节省了很多输入字符的工作或者是使用这个额外的using指令。后面的主意就是这个操作了某些类型的函数常常与这些被操作的类型定义在同样的命名空间中。总的来说，把函数和这个函数操作的类型放在同一个命名空间中也是一个好的做法。

最后一个关于命名空间的有用点就是在using声明中重载函数的名字匹配功能。考虑如下的这个例子：

```
namespace mylib {
   void foo(int);
   void foo(double);
   void foo(std::string);
   // More overloads of foo()...
}

// In some other file...
using mylib::foo; // Which one does this use?
```

这个using声明匹配所有的foo重载函数，因此你不需要给每一个foo的重载函数都写一个这样的using声明。这样做的另一个好处就是如果加入foo的一个重载函数的

www.TopSage.com

大家网 TopSage.com

话，任何使用这个声明如using mylib::foo的代码都是自动加入的，因为这个using声明将会把它们挑选出来（当然是在这个using声明被编译时）。

但是你还是需要明智地使用命名空间，否则你可能会得到一些意外的编译错误，或者你可能给那些使用你的代码的人创建它们。下面是当你需要使用命名空间时的一些流行的指导意见：

保守地使用using namespace xxx

就像前面我解释的那样，引入一个完整的命名空间增加了名字冲突的可能，或者是现在或者是将来的某个时候（别人可能会增加你正在使用的命名空间，从而在你的代码中产生名字冲突）。并且这样也降低了由命名空间提供的模块性。

在头文件中不要使用using语句

头文件被很多其他的文件所包含，因此如果你在一个头文件中使用一个命名空间或者命名空间中的东西时，你就给任何包含这个头文件的文件暴露了你正在使用的任何东西。解决这个问题的办法就是在这个头文件中使用完整的名字路径来限制它。

在#include指令前面不要使用using声明或者定义

如果你这样做的话，在这个头文件中你就会暴露你正在使用的任何东西，这肯定不是这个头文件作者的本意。

如果你遵循这些指导意见的话，那么在一个新的项目中或者给一个现有的项目增加命名空间都会非常容易。

2.5 包含一个内联文件

问题

你有很多需要变成内联函数的成员函数或者独立函数，但是你又不想在头文件中类定义（或者是它们的定义后面）部分定义它们。通过这种办法，你就能保证声明和实现相分离。

解决方案

创建一个*.inl*文件并且在你的头文件的尾部使用#include来包含它。这样完成的功能和你把函数的定义放在头文件的尾部是等价的，但是这样能够让你使得声明和定义相分离。请看示例2-6是如何做的。

示例2-6 使用内联文件

```
// Value.h
#ifndef VALUE_H_ _
#define VALUE_H_ _

#include <string>
```

www.TopSage.com

```
class Value {
public:
   Value (const std::string& val) : val_(val) {}
   std::string getVal() const;
private:
   std::string val_;
};

#include "Value.inl"

#endif VALUE_H_ _

// Value.inl
inline std::string Value::getVal() const {
   return(val_);
}
```

这个解决方案不需要更多的解释。#include语句将会被它的参数的内容代替，因此这里发生的事情就是*Value.inl*文件的内容将会被带入到这个头文件中。因此任何包含这个头文件的文件都将拥有这些内联函数的定义，这样你就不必使你的类的声明变得很散乱。

www.TopSage.com

第3章

数值计算

3.0 概述

即使你不准备编写科学或工程应用程序，你通常也得在一定程度上与数值打交道。本章介绍了在使用C++数值时一些常见问题的解决方案。

其中一些技巧介绍了将不同格式（十六进制、浮点数或科学计数）的数值转换为字符串，或将字符串转换为数值。如果让你自己来编写实现这种转换的代码，是很繁琐的，因此，这里给出了利用标准库或Boost库来使这项工作变得容易的方法。还有一些技巧只是对数值进行操作，例如，数值之间的安全转换、在限定范围内比较两个浮点数、查找某些数值中的最小值和最大值。

本章介绍了一些你在C++中处理数值时可能会遇到的常见问题的解决方案，而没有试图去解决某些特定领域中的问题。如果你正在编写科学或工程应用程序，你还应阅读一下第11章，该章介绍了很多常见的科学与数值算法的技巧。

3.1 将字符串转换为数值

问题

如何将字符串形式的数值转换为诸如`int`或`float`之类的数值类型。

解决方案

有两种方法来实现：使用标准库函数或Boost库（由Kevlin Henney编写）的`lexical_cast`类。标准库函数比较笨拙且不安全，但它们是标准的方法，在某些情况下，你可能需要它们，因此，我作为第一个解决方案来介绍它们。`lexical_cast`则更安全，使用更容易，也更有趣，因此在后面的讨论进行了介绍。

www.TopSage.com

定义在<cstdlib>中的strtol、strtod和strtoul函数可将以null结尾的字符串转换为long int、double或unsigned long类型的数值。你可以用它们把任意进制的字符串转换为数值。示例3-1的代码演示了使用hex2int函数将十六进制的字符串转换为long类型的数值。

示例3-1 将字符串转换为数值

```
#include <iostream>
#include <string>
#include <cstdlib>

using namespace std;

long hex2int(const string& hexStr) {
   char *offset;
   if (hexStr.length() > 2) {
 if (hexStr[0] == '0' && hexStr[1] == 'x') {
    return strtol(hexStr.c_str(), &offset, 0);
 }
   }
   return strtol(hexStr.c_str(), &offset, 16);
}

int main() {
   string str1 = "0x12AB";
   cout << hex2int(str1) << endl;
   string str2 = "12AB";
   cout << hex2int(str2) << endl;
   string str3 = "QAFG";
   cout << hex2int(str3) << endl;
}
```

下面是该程序的输出：

```
4779
4779
0
```

前两个字符串都是包含了12AB的十六进制数值。第一个含有前缀0x，而第二个则没有。第三个不是一个合法的十六进制数，因此函数返回0。

讨论

某些人可能更愿意编写自己的函数，来将十六进制数转换为整数。但为什么要重复做别人已经做过的事情呢？标准库已经提供了这些功能。示例3-1提供了一个封装函数，你只需调用strtol即可。strtol函数实际上是一个来自C标准库的旧函数，它需要传递一个指向以null结尾的字符串的指针，以及另一个字符指针的地址，后面这个指针接收解析结束的地址。但是，在C++中，大多数人更愿意使用功能更强大的字符串类，而不是这个老式的字符指针。因此，hex2int函数带有一个字符串参数。

strtol函数有一点奇怪之处，那就是它允许你使用两种不同的方法来指定基数16。你可以把16作为第三个参数传递给该函数，或者在字符串的前面添加字符0x（就像你在代码中指定十六进制常量一样，但是记住，这里是strtol，你要传递的是一个字符串）。

www.TopSage.com

示例3-1允许你使用任一种方法。如果你传递的是诸如0x12AB之类的字符串，该函数将检测到0x，并把它传递给strtol，其中0将作为第三个参数。否则，把该字符串传递给函数，其中16作为第三个参数。

strtol和strtoul都可以以这种方式工作，唯一不同的是返回类型。strtod也类似，但它不允许你指定基值。

这些旧版本的C函数不是用于将字符串转换为数值的唯一方法。Boost项目提供了一个转换类lexical_cast，它可以将基数为10的数值字符串转换为数值。示例3-2显示了如何使用lexical_cast函数。

示例3-2 使用lexical_cast函数

```
#include <iostream>
#include <string>
#include <boost/lexical_cast.hpp>

using namespace std;

int main() {
   string str1 = "750";
   string str2 = "2.71";
   string str3 = "0x7FFF";
   try {
 cout << boost::lexical_cast<int>(str1) << endl;
 cout << boost::lexical_cast<double>(str2) << endl;
 cout << boost::lexical_cast<int>(str3) << endl;
   }
   catch (boost::bad_lexical_cast& e) {
 cerr << "Bad cast: " << e.what() << endl;
   }
}
```

示例3-2的输出如下：

```
750
2.71
Bad cast: bad lexical cast: source type value could not be
interpreted as target
```

可以看到，对最后一个数值，程序抛出了一个异常。如果需要转换的数值不是十进制的，则需要使用strtlo函数。

strtol函数也有用于宽字符的版本。strtol相应的宽字符函数是wcstol，它定义在<cwchar>中。strtod和strtoul相应的宽字符函数是wcstod和wcstoul。这些函数的使用方法是一样的，只不过把宽字符函数中的参数类型换成了wchar_t*。

参见3.2节。

3.2 将数值转换为字符串

问题

如何将数值（int和float类型）转换为某种格式的字符串。

www.TopSage.com

大家网 TopSage.com

解决方案

有很多种方法可以完成这项工作，但每种方法都有利有弊。这里介绍的第一个技术是使用stringstream类来存储字符串数据，因为它是标准库的一部分，很容易使用。这种方法的使用见示例3-3所示。在后面的讨论中介绍了其他技术。

示例3-3 把一个数值格式化为字符串

```
#include <iostream>
#include <iomanip>
#include <string>
#include <sstream>

using namespace std;

int main() {

   stringstream ss;

   ss << "There are " << 9 << " apples in my cart.";
   cout << ss.str() << endl;  // stringstream::str() returns a string
// with the contents

   ss.str("");    // Empty the string
   ss << showbase << hex << 16;  // Show the base in hexadecimal
   cout << "ss = " << ss.str() << endl;

   ss.str("");
   ss << 3.14;
   cout << "ss = " << ss.str(  ) << endl;
}
```

示例3-3的输出如下：

```
There are 9 apples in my cart.
ss = 0x10
ss = 3.14
```

讨论

stringstream是一种把数据转换成字符串的简便方法，因为它允许使用由标准输入输出流类提供的格式化工具。在示例3-3这种最简单的情况下，使用的是左移位操作符(<<)来将文本和数值数据写入到字符串流中的：

```
ss << "There are " << 9 << " apples in my cart.";
```

操作符<<重载了内置的类型，以相应地格式化输入。当你需要获得保存数据的字符串时，可以使用它的str成员函数：

```
cout << ss.str() << endl;
```

在<iomanip>中有很多的流操作函数，你可以使用它们来实现数值数据的各种格式化。在示例3-3中，我使用了showbase和hex格式化数值。当然还有很多函数。例如，你可以设置显示的精度，而不是使用默认的位数：

```
ss << setprecision(6) << 3.14285;
```

www.TopSage.com

使用操作函数不是最直观的，这就是为什么对于这个主题还有一个技巧。详细信息请参见10.2节。

当然，还有另外一种方法。Boost格式化库（由Samuel Krempp编写）含有一个format类，它可以非常容易地进行格式化和转换。示例3-4显示了如何进行这种转换。

示例3-4 把整数格式化为十六进制

```
#include <iostream>
#include <boost/format.hpp>

using namespace std;
using boost::format;
using boost::io::str;
using boost::io::format_error;

int main( ) {

   try {
 format f("There are %1% ways %2% %3% %4%");

 f % 3;
 f % "to" % "do" % "this.";

 cout << f << endl;

 f.clear(); // Clear buffers to format something else

 f.parse("Those cost $%d.");
 f % 50;

 cout << f << endl;

 int x = 11256099;

 string strx = str(format("%x") % x);
 cout << strx << endl;
   }
   catch (format_error &e) {
 cout << e.what() << endl;
   }
}
```

运行该程序的输出如下：

```
There are 3 ways to do this.
Those cost $50.
abc123
```

使用format类包含两个步骤，首先创建一个format对象，然后给它发送内容。在示例3-4中，我使用最简单的语法形式创建了一个format对象：

```
format f("There are %1% ways %2% %3% %4%");
```

在format的字符串中，占位符是左右两边都带有一个%的数值。然后，我给format发送如下内容：

```
f % 3;
f % "to" % "do" % "this.";
```

在格式化库中，操作符%被重载了，以便添加变量到format对象的左边。你可以每行

www.TopSage.com

使用它一次，或在一行中多次调用它。它类似于数据流的操作符<<。说到操作符<<，它也被重载了，这样你就可以直接把格式化对象写入到输出类中。另外，如果你需要把结果保存到字符串中，可以使用str成员函数：

```
string s = f.str();
```

如果你是一个喜欢使用printf的人，可以使用format的printf指示符：

```
f.parse("Those cost $%d.");
f % 50;
```

如果给format函数提供的内容变量太多或太少，并要把它写入到数据流或输出一个格式化的字符串，就会抛出一个format_error（或相应的子类）异常。

format类的功能很强大，它有太多优雅的格式化性能值得我们去了解。从Boost的网址*www.boost.org*可以下载Boost或阅读其文档。

你也可以使用sprintf或相关的函数来把数值转换为字符串。通常，你应该避免这样做，因为这是不安全的，有比这更好的实现方法。

参见第10章。

3.3 测试字符串中是否含有有效的数值

问题

如何从一个字符串中找出它是否含有有效的数值？

解决方案

你可以使用Boost库的lexical_cast函数模板来测试有效数值。使用这种方法，一个有效的数值可以包含一个减号或加号，但不能有空格。在示例3-5中给出了一些使用lexical_cast的格式化示例。

示例3-5 验证字符串数值

```
#include <iostream>
#include <boost/lexical_cast.hpp>

using namespace std;
using boost::lexical_cast;
using boost::bad_lexical_cast;

template<typename T>
bool isValid(const string& num) {

   bool res = true;

   try {
 T tmp = lexical_cast<T>(num);
   }
   catch (bad_lexical_cast &e) {
 res = false;
```

www.TopSage.com

```
    }

    return(res);
}

void test(const string& s) {

    if (isValid<int>(s))
  cout << s << " is a valid integer." << endl;
    else
  cout << s << " is NOT a valid integer." << endl;

    if (isValid<double>(s))
  cout << s << " is a valid double." << endl;
    else
  cout << s << " is NOT a valid double." << endl;

    if (isValid<float>(s))
  cout << s << " is a valid float." << endl;
    else
  cout << s << " is NOT a valid float." << endl;
}

int main() {

    test("12345");
    test("1.23456");
    test("-1.23456");
    test(" - 1.23456");
    test("+1.23456");
    test("  1.23456  ");
    test("asdf");
}
```

下面是该程序的输出：

```
12345 is a valid integer.
12345 is a valid double.
12345 is a valid float.
1.23456 is NOT a valid integer.
1.23456 is a valid double.
1.23456 is a valid float.
-1.23456 is NOT a valid integer.
-1.23456 is a valid double.
-1.23456 is a valid float.
 - 1.23456 is NOT a valid integer.
 - 1.23456 is NOT a valid double.
 - 1.23456 is NOT a valid float.
+1.23456 is NOT a valid integer.
+1.23456 is a valid double.
+1.23456 is a valid float.
  1.23456   is NOT a valid integer.
  1.23456   is NOT a valid double.
  1.23456   is NOT a valid float.
asdf is NOT a valid integer.
asdf is NOT a valid double.
asdf is NOT a valid float.
```

讨论

lexical_cast 函数模板可以把一个值从一种类型转换为另一种类型。其声明如下：

```
template<typename Target, typename Source>
Target lexical_cast(Source arg)
```

www.TopSage.com

Source是初始变量的类型，Target要转换成的变量类型。因此，如果你要从字符串转换为整型，可以如下调用lexical_cast：

```
int i = lexical_cast<int>(str); // str is a string
```

lexical_cast进行解析和转换。如果不能进行转换，它将抛出一个bad_lexical_cast异常。在示例3-5中，我只是想测试有效性，并不需要保留目标变量，因此，如果没有抛出异常就返回true，否则返回false。

你只需为lexical_cast提供第一个模板参数，因为它是一个模板函数，这就意味着编译器可以推导出函数参数的类型，并把它作为第二个模板参数。要说明这种特性的最好方法是举例，因此，让我们来看一个代码示例。你可以不是像前面的代码中那样调用lexical_cast，而是：

```
int i = lexical_cast<int, string>(str);
```

这样的结果是一样的，但你并不需要提供字符串参数，因为编译器知道str是一个字符串，并且可以计算出其余的内容。

如果你想编写类似的封装函数来测试有效性，且只返回true或false，把它编写为一个函数模板就可以了。这样，你就只需编写一次参数化的类型，以后，每次都会根据你使用的不同类型实例化成不同的版本。

lexical_cast也可以很方便地把数值类型转换为其他类型，我将在3.6节中详细介绍。

参见3.6节。

3.4 在限定精度内进行浮点数的比较

问题

你需要比较两个浮点数，但你只是想测试在有限的位数内是相等、大于或小于。例如，如果精度为0.0001，那么你希望3.33333和3.33333333是相等的。

解决方案

可以编写你自己的比较函数，它带有一个限定比较操作的精度的参数。示例3-6显示了这种比较函数的基本技术。

示例3-6 浮点数的比较

```
#include <iostream>
#include <cmath>    // for fabs()

using namespace std;

bool doubleEquals(double left, double right, double epsilon) {
   return (fabs(left - right) < epsilon);
}
```

www.TopSage.com

大家网 TopSage.com

```
bool doubleLess(double left, double right, double epsilon,
 bool orequal = false) {
   if (fabs(left - right) < epsilon) {
 // Within epsilon, so considered equal
 return (orequal);
   }
   return (left < right);
}

bool doubleGreater(double left, double right, double epsilon,
    bool orequal = false) {
   if (fabs(left - right) < epsilon) {
 // Within epsilon, so considered equal
 return (orequal);
   }
   return (left > right);
}

int main() {

   double first = 0.33333333;
   double second = 1.0 / 3.0;
   cout << first << endl;
   cout << second << endl;

   // Straight equalify test. Fails when you wouldn't want it to.
   // (boolalpha prints booleans as "true" or "false")
   cout << boolalpha << (first == second) << endl;
   // New equality. Passes as scientific app probably wants.
   cout << doubleEquals(first, second, .0001) << endl;
   // New less-than
   cout << doubleLess(first, second, .0001) << endl;
   // New Greater-than
   cout << doubleGreater(first, second, .0001) << endl;
   // New less-than-or-equal-to
   cout << doubleLess(first, second, .0001, true) << endl;
   // New greater-than-or-equal-to
   cout << doubleGreater(first, second, .0001, true) << endl;
}
```

下面是该程序的输出：

```
0.333333
0.333333
false
true
false
false
true
true
```

讨论

示例3-6的代码开始有两个值，即0.33333333和1.0 / 3.0的计算结果。使用cout的默认格式化形式输出这两个值，将显示相同的结果0.333333。但是，当你比较这两个值时，它们的确是不同的。1.0 / 3.0的有效位数比0.33333333的多，因此，根据你的计算机，这两个数值是不相等的。但是，在某些应用程序中，你可能希望这两个数值相等。

处理这个问题的方法是自己编写三个用于比较两个数值的函数：`doubleLess`、`doubleEquals`和`doubleGreater`，每个函数都以两个数值作为其参数。另外，

doubleLess和doubleGreater还有一个参数，当函数为true时，使函数按小于等于或大于等于分别执行。

要使这些函数能处理精度，首先考虑doubleEquals函数。该函数不是测试是否相等，而是测试这两个数值之差是否在用户指定的范围之内（上面示例使用的是0.0001）。如果是，那么函数返回true，说明数值的确相等。这样，数值0.3333、0.33333、0.333333、0.33333333333和0.33333323438都将是相等的。

要处理小于或大于操作，首先测试数值在指定范围内是否相等，就像在doubleEquals所做的那样。如果相等，且你希望在测试中包括相等性，则返回true，否则返回false。如果不相等，再进行直接比较。

3.5 解析含有用科学计数法表示的数值的字符串

问题

如何把用科学计数法表示的数值字符串存储到double变量中？

解决方案

要解析用科学计数法表示的数值，最直接的方法是使用C++函数库的声明在<sstream>中的内置stringstream类，见示例3-7。

示例3-7 解析用科学计数法表示的数值

```
#include <iostream>
#include <sstream>
#include <string>

using namespace std;

double sciToDub(const string& str) {

   stringstream ss(str);
   double d = 0;
   ss >> d;

   if (ss.fail()) {
 string s = "Unable to format ";
 s += str;
 s += " as a number!";
 throw (s);
   }

   return (d);
}

int main() {

   try {
 cout << sciToDub("1.234e5") << endl;
 cout << sciToDub("6.02e-2") << endl;
 cout << sciToDub("asdf") << endl;
   }
```

www.TopSage.com

```
  catch (string& e) {
cerr << "Whoops: " << e << endl;
  }

  }
```

下面是该代码的输出：

```
123400
0.0602
Whoops: Unable to format asdf as a number!
```

讨论

不用奇怪，stringstream是一个字符串，其行为与数据流类似。它声明在<string>中。如果你需要解析一个含有用科学计数法表示的数值的字符串（参见3.2节），stringstream可以很好地完成这项工作。标准的数据流类已“知道”了如何解析数值，因此如果不是必须，否则不要浪费时间去重新实现这些代码。

在示例3-7中，我编写了一个简单的函数sciToDub，它带有一个字符串参数，返回该字符串所包含的double值。在sciToDub中，我是如下使用stringstream的：

```
stringstream ss(str); // Construct from a string
double d = 0;
ss >> d;

if (ss.fail()) {
   string s = "Unable to format ";
   s += str;
   s += " as a number!";
   throw (s);
}
return (d);
```

这里最重要的部分是，你必须做的一切就是使用右移位操作符(>>)，把字符串流读入到double变量中，就像从cin中读取数据一样。

对了，这不是你要做的全部工作。如果stringstream中的某个数值不能写入>>操作符右边的变量中，就要在数据流中设置一个失败位。使用fail成员函数（这其实是basic_ios的成员函数，basic_ios是stringstream的超类）可以检测这个位。另外，如果操作失败，>>操作符右边的变量不会发生改变。

但是，为了能够通用，你可以避免为int、float、double和任何你想通过编写函数模板来进行转换的其他类型分别编写不同的sciToDub版本。请看下面的新版本：

```
template<typename T>
T strToNum(const string& str) {
   stringstream ss(str);
   T tmp;
   ss >> tmp;

   if (ss.fail()) {
 string s = "Unable to format ";
 s += str;
 s += " as a number!";
 throw (s);
```

www.TopSage.com

```
    }

    return (tmp);
}
```

现在，如果你想把字符串转换为数值类型，就可以这样来做：

```
double d = strToNum<double>("7.0");
float f = strToNum<float>("7.0");
int i = strToNum<int>("7.0");
```

你也可以把字符类型作为模板参数，这样做是很简单的，因此留给读者作为练习。

参见3.2节。

3.6 不同数值类型之间的转换

问题

如何把一种类型的数值转换为另一种类型？例如，把int类型转换为short类型，或者把short类型转换为int类型，且想捕获任何运行时的上溢或下溢错误。

解决方案

使用Boost函数库的numeric_cast类模板。它可进行运行时检测，如果输入一个数值时，变量发生上溢或下溢，将抛出一个bad_numeric_cast类型的异常。示例3-8显示了如何实现。

示例3-8 安全的数值转换

```
#include <iostream>
#include <boost/cast.hpp>

using namespace std;
using boost::numeric_cast;
using boost::bad_numeric_cast;

int main() {

   // Integer sizes
   try {
 int i = 32767;
 short s = numeric_cast<short>(i);

 cout << "s = " << s << endl;

 i++; // Now i is out of range (if sizeof(short) is 2)
 s = numeric_cast<short>(i);
   }
   catch (bad_numeric_cast& e) {
 cerr << e.what() << endl;
   }

   try {
 int i = 300;
 unsigned int ui = numeric_cast<unsigned int>(i);

 cout << ui << endl; // Fine
```

www.TopSage.com

```
 i *= -1;
 ui = numeric_cast<unsigned int>(i); // i is negative!
   }
   catch (bad_numeric_cast& e) {
 cerr << e.what() << endl;
   }

   try {
 double d = 3.14;
 int i = numeric_cast<int>(d);

 i = numeric_cast<int>(d); // This shaves off the 0.14!

 cout << i << endl;  // i = 3

   }
   catch (bad_numeric_cast& e) {
 cerr << e.what() << endl;
   }
}
```

讨论

你可能知道，基本的C++数据类型有不同的大小。C++标准为各种数据类型指定了严格的相对大小——int总是至少与short int一般大，但没有指定绝对大小。这就意味着，如果你要把long int类型的数据保存到short int，或把int保存到unsigned int中，那么你就有可能会使源变量的一些信息丢失，如符号甚至是部分数值。

只知道这会引起问题是不够的。你的存储空间可能有限，你需要存储最小的可能数值。为了安全起见，可以使用Boost库的numeric_cast在运行时来捕获数据的丢失情况

numeric_cast的语法很简单。它是一个函数模板，声明如下：

```
template<typename Target, typename Source>
inline Target numeric_cast(Source arg)
```

如果你已经阅读过3.1节或3.3节就会知道，它很像lexical_cast。它有两个参数Target和Source，分别表示原始值和目标值的类型。因为它是函数模板，编译器可以推导出源模板参数的类型，因此，你只需提供Target的类型，具体如下：

```
int i = 32767;
short s = numeric_cast<short>(i);
```

short是Target参数的模板变量。编译器可以计算出Source是int类型的，因为i是int类型的。

在这种情况下，我要把int数据填入到short中。在我的操作系统中（Windows XP），int为4字节，而short为两字节。short是有符号的，这意味着只有15位可用于表示数值，因此，它能保存的最大正数值为32 767。上面的代码片断可以无故障地运行，但当我使i递增1时，将超出short的范围：

```
i++;
s = numeric_cast<short>(i); // Uh-oh
```

www.TopSage.com

大家网 TopSage.com

于是将抛出一个 bad_numeric_cast 异常 —— 你已经有这种概念了。看看示例 3-8 的其余部分，如果你试图把一个带负号的数值赋给无符号的数据类型，numeric_cast也可以捕获到下溢异常。

但numeric_cast并没有解决所有问题。例如，如果你把一个浮点数值保存到非浮点数据类型中，小数点右边的内容将全部丢失，对吗？对此，numeric_cast不能帮助你，因此，不要认为它把你从所有风险中解救出来了。例如，请看来自示例 3-8 的代码片断：

```
double d = 3.14;
int i = numeric_cast<int>(d);  // Ouch
```

这里不会抛出异常。但如果你试图如下做：

```
double d = -3.14;
unsigned int ui = numeric_cast<unsigned in>(d);
```

因为这不仅会丢弃小数点右边的所有内容，而且还丢失了负号，这是很糟糕的事情。

参见 3.1 节和 3.3 节。

3.7 获得某种数值类型的最小和最大值

问题

对于某个数值类型，例如 int 或 double，如何知道你的平台对这种类型的数值所能表示的最大或最小值？

解决方案

使用<limits>中的numeric_limits类模板可以知道某个数值类型所能表示的最大和最小值，参见示例 3-9。

示例 3-9 获得数值的极值

```
#include <iostream>
#include <limits>

using namespace std;

template<typename T>
void showMinMax() {
   cout << "min: " << numeric_limits<T>::min() << endl;
   cout << "max: " << numeric_limits<T>::max() << endl;
   cout << endl;
}

int main() {

   cout << "short:" << endl;
   showMinMax<short>();
   cout << "int:" << endl;
   showMinMax<int>();
   cout << "long:" << endl;
```

www.TopSage.com

```
    showMinMax<long>();
    cout << "float:" << endl;
    showMinMax<float>();
    cout << "double:" << endl;
    showMinMax<double>();
    cout << "long double:" << endl;
    showMinMax<long double>();
    cout << "unsigned short:" << endl;
    showMinMax<unsigned short>();
    cout << "unsigned int:" << endl;
    showMinMax<unsigned int>();
    cout << "unsigned long:" << endl;
    showMinMax<unsigned long>();
}
```

下面是在Windows XP系统中的Visual C++ 7.1平台上的输出：

```
short:
min: -32768
max: 32767

int:
min: -2147483648
max: 2147483647

long:
min: -2147483648
max: 2147483647

float:
min: 1.17549e-038
max: 3.40282e+038

double:
min: 2.22507e-308
max: 1.79769e+308

long double:
min: 2.22507e-308
max: 1.79769e+308

unsigned short:
min: 0
max: 65535

unsigned int:
min: 0
max: 4294967295

unsigned long:
min: 0
max: 4294967295
```

讨论

示例3-9显示了一个简单的示例，用于获得某个数值类型的最小和最大值。numeric_limits模板类对所有内置的数据类型(包括数值和非数值类型)都有一个特殊说明。标准库说明文件不仅包括了示例3-9中所使用的所有数据类型，而且还包括以下类型：

```
bool
char
signed char
```

www.TopSage.com

```
unsigned char
wchar_t
```

min和max是numeric_limits的静态成员函数，分别为你所传递的数据类型返回最大和最小值。

www.TopSage.com

第4章

字符串与文本

4.0 概述

本章介绍操作字符串与文本文件的一些技巧。大多数C++程序（不管它们是哪种应用）都会在一定程度上需要操作字符串和文本文件。但是，尽管应用不同，要求却是相同的，对字符串需要裁剪、填充、查找、分割等等；对文本文件需要封装、重新格式化、以及读出定界文件等等。本章介绍的这些技巧为很多在C++标准库中没有现成解决方案的常见需求提供了解决办法。

标准库是可移植、标准化的，通常，其效率至少不会低于自己编写的解决方案，因此，在后面的示例中，我更喜欢使用标准库函数，而不是自己从头编写代码。有一个丰富的框架用于操作和管理字符串和文本：很多是在basic_string类模块中，对字符串而言是basic_istream和basic_ostream，对输入输出文本流是basic_istream和basic_ostream。本章中几乎所有的技术都是使用或扩展这些类模板。如果没有我所需要的，就使用标准库中的通用、预先构建好的解决方案，即算法和容器。

每个人都使用字符串，因此，如果你所需要的在标准库中没有，或许其他人已编写了。由Pavol Droba编写的Boost字符串算法库填补了标准库的很多空白，它实现了你经常需要使用的大多数算法，而且是可移植的，效率也很高。从*www.boost.org*可以了解字符串算法库的详细信息和文档。在字符串算法库与本章所介绍的解决方案之间有一定的重叠。在大多数情况下，对相关的解决方案，我都会提供Boost算法的示例（或者至少是介绍说明）。

对于大多数的示例，我都提供了无模板和有模板两个版本。我这样做有两个原因。首先，使用字符数据的标准库大部分是类或函数模板，其参数为char或wchar_t类型的字符。在这种模式下，你需要知道这种标准库软件的最大容量。其次，不管你是否使用标准库，

www.TopSage.com

类和函数模板为编写通用软件提供了很好的性能。但是，如果你不需要模板，你可以使用无模板的版本。尽管这样，如果你对它们比较生疏，我还是建议你多练习模板的使用。

标准库大量使用了模板，并使用typedef指令来使程序开发人员从模板使用的冗长语法中解脱出来。因此，我交替地使用了basic_string、string和wstring，因为如果某一个能用，通常也就都可以使用。string和wstring也就是basic_string <char>和basic_string<wchar_t>。

最后，你可能会注意到，本章没有一个技巧使用了C风格的字符串，也就是说，以null结尾的字符串数组。标准库对C++字符串提供了高效且可扩展的支持，比使用C风格的字符串函数（标准库之所以提供它们，主要是为了向后兼容）有更好的灵活性、安全性和通用性，这就是C++字符串类。

4.1 填充字符串

问题

如何把某些字符串反复地填充成特定的宽度？例如，你可能需要把“Chapter 1”反复地填充成20个字符宽，从而形成“Chapter 1.....”。

解决方案

使用字符串的插入或附加成员函数，可以把字符串插入到开头或末尾。例如，要用字符X把字符串填充成20个字符：

```
std::string s = "foo";
s.append(20 - s.length(), 'X');
```

如果要在开头插入，可以如下进行：

```
s.insert(s.begin(), 20 - s.length(), 'X');
```

讨论

这两个函数在使用上的区别是insert的第一个参数。它是一个迭代器，指向要插入位置的右边的字符。begin成员函数返回指向字符串的第一个元素的迭代器，因此，在上面的示例中，字符系列被插入到了左边。这两个函数相同的参数是重复插入字符的次数和字符本身。

insert和append实际上是<string>头文件中basic_string类模板的成员函数（string是basic_string<char>的类型定义，wstring是basic_string<wchar_t>的类型定义），因此，它们可以使用窄字符和宽字符。就像上面的示例那样，按需使用它们，可以很好地工作。但是，如果在你自己的函数中使用basic_string成

www.TopSage.com

大家网 TopSage.com

员函数，就需要构建在标准库的已有通用设计之上，使用函数模板。请看示例4-1的代码，它定义了一个通用的pad函数模板，使用的是basic_string类型的字符。

示例4-1 通用的pad函数模板

```
#include <string>
#include <iostream>

using namespace std;

// The generic approach
template<typename T>
void pad(basic_string<T>& s,
    typename basic_string<T>::size_type n, T c) {
   if (n > s.length())
 s.append(n - s.length(), c);
}

int main() {

   string  s  = "Appendix A";
   wstring ws = L"Acknowledgments"; // The "L" indicates that
  // this is a wide char
   pad(s, 20, '*');    // literal
   pad(ws, 20, L'*');

   // cout  << s  << std::endl;     // You shouldn't be able to
   wcout << ws << std::endl;        // run these at the same time
}
```

示例4-1中的pad函数用字符c把给定字符串s填充成n宽度。由于函数模板为字符串元素使用了一个参数化的类型（T），因此它可以使用任意类型的basic_string字符，即char、wchar_t或其他自定义的字符。

4.2 裁剪字符串

问题

如何从字符串的末尾裁剪掉一定数量的字符（通常是空白符）？

解决方案

使用迭代器来标识要删除的字符串部分，并用erase成员函数来删除它。示例4-2显示了rtrim函数，它从字符串的末尾删除一个字符。

示例4-2 从字符串中删除字符

```
#include <string>
#include <iostream>

// The approach for narrow character strings
void rtrim(std::string& s, char c) {

   if (s.empty())
 return;

   std::string::iterator p;
   for (p = s.end(); p != s.begin() && *--p == c;);
```

www.TopSage.com

```
   if (*p != c)
 p++;

   s.erase(p, s.end());
}

int main()
{
   std::string s = "zoo";

   rtrim(s, 'o');

   std::cout << s << '\n';
}
```

讨论

示例4-2巧妙地处理了字符的串，其实它只能处理字符串。正如你在示例4-1所见到的那样，你也可以利用basic_string的通用设计，并使用一个函数模板。示例4-3使用一个函数模板，从任意类型的字符串末尾删除字符。

示例4-3 rtrim的泛型函数版本

```
#include <string>
#include <iostream>

using namespace std;

// The generic approach for trimming single
// characters from a string
template<typename T>
void rtrim(basic_string<T>& s, T c)
{
   if (s.empty())
 return;

   typename basic_string<T>::iterator p;
   for (p = s.end(); p != s.begin() && *--p == c;);

   if (*p != c)
 p++;

   s.erase(p, s.end());
}

int main() {

   string s = "Great!!!!";
   wstring ws = L"Super!!!!";

   rtrim(s, '!');
   rtrim(ws, L'!');

   cout << s << '\n';
   wcout << ws << L'\n';
}
```

该函数与示例4-2的工作完全一样，但由于它参数化为所使用的字符类型，因此可以使用任意类型的basic_string。

示例4-2和示例4-3是从字符串删除一个字符。但是，删除空白符就要困难一些，因为空白符可能是多个字符中的一个。为此，标准库提供了一种简洁的方法，这就是<cctype>

www.TopSage.com

头文件中的isspace函数（wchar_t等同于<cwctype>头文件中的iswspace）。示例4-4定义了一个用于删除末尾空白符的通用函数。

示例4-4 删除末尾空白符

```
#include <string>
#include <iostream>
#include <cctype>
#include <cwctype>

using namespace std;

template<typename T, typename F>
void rtrimws(basic_string<T>& s, F f) {

   if (s.empty())
 return;

   typename basic_string<T>::iterator p;
   for (p = s.end(); p != s.begin() && f(*--p););

   if (!f(*p))
 p++;

   s.erase(p, s.end());
}

// Overloads to make cleaner calling for client code
void rtrimws(string& s) {
   rtrimws(s, isspace);
}

void rtrimws(wstring& ws) {
   rtrimws(ws, iswspace);
}

int main() {

   string s = "zing    ";
   wstring ws = L"zong    ";

   rtrimws(s);
   rtrimws(ws);

   cout << s << "|\n";
   wcout << ws << L"|\n";
}
```

示例4-4的函数模板rtrimws是一个通用的函数模板，与前面几个示例类似，它接收一个basic_string类型的参数，删除该参数后面的空白符。与其他示例不同的是，它采用的是一个函数对象，而不是字符，用来测试字符串的元素，以确定是否要删除它。

你不需要像上面示例那样去重载rtrimws函数，但这样可以使得使用该函数时语法更清楚，因为调用代码可以忽略谓词函数参数。

但这种解决方案需要你自己编写代码。如果你更愿意使用类库的话，Boost字符串算法类库就是一个不错的选择，这个类库为裁剪字符串提供了许多函数，使用这个类库，你所需要的都能在这个类库中找到。事实上，这个字符串算法类库有很多便利的裁剪函数，如果你能使用这个类库的话，应该好好看看。表4-1中列出了这个类库中的很多裁剪函数模

www.TopSage.com

板。因为这些函数只是模板函数，所以它们带有代表不同类型的模板参数。下面就是它们每个所代表的含义：

- Seq：这个类型代表C++标准中的序列。
- Coll：集合类型，它比标准的序列类型有更少的限制。请参考Boost字符算法定义，有关于集合需求的详细描述。
- Pred：是个函数对象或函数指针，它带有单一的参数并且返回一个布尔值。换句话说，就是一个一元断言。你可以给一些裁剪函数提供你自己的一元断言从而裁剪掉一些符合一定标准的元素。
- OutIt：C++标准中的输出迭代器。你可以增加它或给它指定新的位置，从而可以给它指向的序列末尾增加元素。

表4-1 Boost字符串裁剪函数模板

声明	描述
`template<typename Seq>` `void trim(Seq& s,` `  const locale& loc =` `  locale());`	使用语言环境分类函数来标识空格字符从而裁剪掉序列s两端的空格字符
`template<typename Seq,` ` typename Pred>`	一元断言p(*it)为真的情况下，从序列s的每一端开始裁剪掉元素，一元断言p(*it)中的it是一个迭代器，指向序列中的每个元素。当一元断言p(*it)为假时，裁剪过程停止
`template<typename Seq>` `Seq trim_copy(const Seq& s,` `  const locale& loc =` `  locale());`	与trim函数具有相同的功能，但它并不修改序列s，而是返回一个具有裁剪后结果的新序列
`template<typename Seq,` `  typename Pred>` `Seq trim_copy_if(const Seq& s,` `  Pred p);`	与trim_if函数具有相同的功能，但它并不修改序列s，而是返回一个具有裁剪后结果的新序列
`template<typename OutIt,` ` typename Coll,` ` typename Pred>` `OutIt trim_copy_if(OutIt out,` `  const Coll& c, Pred p);`	这个函数和前面那个版本的trim_copy_if函数在功能上基本相同，但还有些区别。首先，它有健壮的异常安全性检查。第二，它带有一个输出迭代器作为第一个参数，并返回一个输出迭代器，这个返回输出迭代器指向目标序列末端的位置。最后，它带有一个集合类型而不是序列类型
trim_left trim_right	与trim在工作方式上相同，但仅仅是裁剪掉字符串左边或右边的字符

www.TopSage.com

表4-1 Boost字符串裁剪函数模板（续）

声明	描述
trim_left_if trim_right_if	与trim_if在工作方式上相同，但仅仅是裁剪掉字符串左边或右边的字符
trim_left_copy trim_right_copy	与trim_copy在工作方式上相同，但仅仅是裁剪掉字符串左边或右边的字符
trim_left_copy_if trim_right_copy_if	与trim_copy_if在工作方式上相同，但仅仅是裁剪掉字符串左边或右边的字符。每一对都有两个版本，一个版本是在序列上操作，另一个是在集合上操作

表4-1中最先描述的那四个函数模板是这个字符串算法类库中裁剪函数的核心。其余的都是这四个的变种。表4-5中给出了一个简单使用这些函数的例子。从中，你能发现使用这些函数优于使用string类成员函数。

示例4-5 使用Boost字符串裁剪函数

```
#include <iostream>
#include <string>
#include <boost/algorithm/string.hpp>

using namespace std;
using namespace boost;

int main() {

   string s1 = " leading spaces?";

   trim_left(s1);                          // Trim the original
   string s2 = trim_left_copy(s1);         // Trim, but leave original intact

   cout << "s1 = " << s1 << endl;
   cout << "s2 = " << s2 << endl;

   s1 = "YYYYboostXXX";
   s2 = trim_copy_if(s1, is_any_of("XY")); // Use a predicate to

   trim_if(s1, is_any_of("XY"));

   cout << "s1 = " << s1 << endl;
   cout << "s2 = " << s2 << endl;

   s1 = "1234 numbers 9876";
   s2 = trim_copy_if(s1, is_digit());

   cout << "s1 = " << s1 << endl;
   cout << "s2 = " << s2 << endl;

   // Nest calls to trim functions if you like
   s1 = "    ****Trim!****";
   s2 = trim_copy_if(trim_copy(s1), is_any_of("*"));

   cout << "s1 = " << s1 << endl;
   cout << "s2 = " << s2 << endl;
}
```

www.TopSage.com

大家网 TopSage.com

示例4-5表明了如何使用Boost字符串裁剪函数。它们都很好理解，所以除了表4-1之外，无需做更多详细的解释。这个例子中不在表4-1中的函数就是is_any_of。这是一个返回一个断言函数对象的函数模板，返回的断言函数对象可用于trim_if风格的函数。当你想裁剪一个字符集合时，就可使用此函数。这个函数与is_from_range具有相同的函数形式，都带有两个参数，并且返回一个一元断言，当这个一元断言为真时，表明这个集合中的某个字符在这个函数的两个参数所指明的范围中。下面的这个例子是从一个字符串中裁剪掉从a到d的字符：

```
s1 = "abcdXXXabcd";
trim_if(s1, is_from_range('a', 'd'));
cout << "s1 = " << s1 << endl;  // Now s1 = XXX
```

请注意，上面的函数工作方式是大小写敏感的，由于从a到d的字符当中不包含大写字符，所以X不在其中。

4.3 将字符串存储到字符系列中

问题

如何存储一个字符串集合到一个序列当中，并且使得这个序列与数组的工作方式一样？

解决方案

使用vector作为你的字符串的类数组存储。示例4-6提供了一个简单的实现。

示例4-6 在vector中存储字符串

```
#include <string>
#include <vector>
#include <iostream>

using namespace std;

int main() {

   vector<string> v;

   string s = "one";
   v.push_back(s);

   s = "two";
   v.push_back(s);

   s = "three";
   v.push_back(s);

   for (int i = 0; i < v.size(); ++i)
   {
 cout << v[i] << '\n';
   }
}
```

vector在随机访问和别的一些方面遵循了数组的语意，所以它们容易使用。然而vector仅仅是标准类库中众多序列当中的一种，下面将要围绕着这个主题展开讨论。

www.TopSage.com

讨论

vector是一个提供了数组风格的随机访问操作符[]并且长度可以动态改变的序列。它的成员函数push_back通过复制构造函数从它的输入参数来构造一个对象，并把这个构造的对象作为最后一个元素加到vector当中，同时给vector的长度加1。成员函数pop_back则相反，删除掉vector中的最后一个元素并使vector长度减1。从vector末端插入和删除元素所花费的时间是恒定的，而且从vector的别的位置插入和删除花费的是线性时间。这些都是vector的基本特性。当然，除了这些基本特性之外，vector还有很多别的特性。

在很多案例中，要想代替C风格的数组，vector应该是你第一选择。首先，它们的长度是动态可变的，这就意味着你可以按需来增长它。你就不用与使用C数组那样，不得不做各种各样的研究来计算出最佳的静态数组长度。vector是按需增长的，你能够根据需要来手工地增大或减小它的长度。第二，vector提供了成员函数（但不是操作符[]）来进行边界检查，因此，你就能用这个成员函数来做一些有用的检查以防止你引用了一个不存在的索引，而不会只能简单地监视你的程序是否崩溃，或者更坏的是使用这些被破坏的数据执行指令。请看示例4-7，它给出了如何处理索引越界功能。

示例4-7 vector的索引越界检查

```
#include <iostream>
#include <vector>
#include <exception>

using namespace std;

int main() {

   char carr[] = {'a', 'b', 'c', 'd', 'e'};

   cout << carr[100000] << '\n'; // Whoops, who knows what's going
   // to happen
   vector<char> v;
   v.push_back('a');
   v.push_back('b');
   v.push_back('c');
   v.push_back('d');
   v.push_back('e');

   try {
 cout << v.at(10000) << '\n';    // at checks bounds and throws
   } catch(out_of_range& e) {    // out_of_range if it's invalid
 cerr << e.what() << '\n';
   }
}
```

如果你捕获住out_of_range异常（定义在<stdexcept>）的话，你就能以一种有意义的方式来处理不合法的索引访问。并且，你可以基于你的实现来调用你自己的成员函数来给出有用的错误信息，就像示例4-7那样返回异常的描述信息：

```
    invalid vector<T> subscript
```

www.TopSage.com

C++中，vector并不是你唯一的选择，还有很多种方式可用来存储序列。除了vector，还有list、set和deque。总的来说，它们具有大部分相同的操作，同时又具有自己独特的操作。此外，它们又有不同的算法复杂度、存储需求和语意特征。所以，在它们当中，也就有不同的选择。

请仔细看示例4-6。你大概注意到在我用push_back把字符串追加到容器末端前不断地在改变字符串s的值。你可能认为比较合理的输出应该是：

```
three
three
three
```

我三次把同一个字符串追加到这个vector的末端，每次我都重新给它赋值，难道这个vector中所有的元素不是引用同样的对象吗？不是，这是STL容器中非常重要的一点。

STL存储的是你加入对象的副本，而不是对象本身。因此，在我把我的三个字符串加到容器后，在内存中，有四个对象：三个在容器中的对象副本和一个已经赋值的对象副本。

因此，如此多的额外副本将使得代价非常大。是的，代价是非常大，因为无论你写得什么东西，如果是使用很多的字符串，你都将为复制它们付出处理器时间或者存储空间，或者两者都有。把元素复制进容器或复制出容器，这是STL的内在行为，并且所有的容器都是以这种方式工作的。

这种问题的一种解决方案（肯定不是有用的方案）就是在容器中存储对象的指针而不是对象本身。请记住当容器被清除时，它并不删除这个指针所指向的对象。你的代码为这个指针分配了内存，因此，你的代码不得不自己去把它清除。这种情况在容器被完全清除或元素被移除时就会发生。

为了提供一种替换方案，让我们考虑另外一种。假设类模板list，它定义在<list>中，它是一个双向链表。如果你打算在这个序列的中间做很多的插入和删除操作，或者你想确保当你修改这个序列时迭代器指向序列中无效的元素，你或许想使用list。示例4-8使用list代替vector来存储一些字符串；同时它不像你在处理简单数组时那样不得不使用索引操作符那样，它使用for_each代替索引操作符来枚举这些字符串并打印出来。

示例4-8 使用list来存储字符串

```
#include <string>
#include <list>
#include <algorithm>
#include <iostream>

using namespace std;

void write(const string& s) {
   cout << s << '\n';
}
```

www.TopSage.com

```
int main() {
   list<string> lst;

   string s = "knife";
   lst.push_front(s);

   s = "fork";
   lst.push_back(s);

   s = "spoon";
   lst.push_back(s);

   // A list doesn't have random access, so
   // use for_each() instead
   for_each(lst.begin(), lst.end(), write);
}
```

这里STL中序列的简短介绍脱离了原来的主题（用序列来存储字符串）。所以这里没有给出这个主题的详细介绍。有关STL的详细信息，请参考*C++ in a Nutshell*（Ray Lischner, O'Reilly）一书的第10章。

4.4 获得字符串的长度

问题

如何获得字符串的长度？

解决方案

使用string的成员函数：

```
std::string s = "Raising Arizona";
int i = s.length();
```

讨论

获取一个字符串的长度是一项微不足道的任务，但它是用来讨论字符串（宽字节和窄字节字符）分配方案的好机会。字符串不像C风格的以null结尾的字符数组，其长度是可动态改变的，并可按需增长。大部分标准类库的实现都以一个比较低的容量开始，当这个容量达到后，容量再增长一倍。知道如何分析它的增长情况，或者更进一步讲是它的精确算法，对于诊断字符串性能问题是有帮助的。

basic_string中的字符存储在一块缓存中，这块缓存是一块具有静态大小的连续内存。字符串使用的这块缓存是初始分配好一定长度的，并且随着字符被加入到这个字符串中，这块缓存被填充直到达到它的初始分配的容量。当这种情况发生时，缓存就依据一定的算法增长。特殊的情况下，分配一块新的更大的缓存，旧的缓存中的字符就被复制到这块新的缓存中，并且这块旧的缓存就被删除掉。

你能够使用capacity成员函数得到缓存的大小（不是缓存所包含的字符的个数，而是它能包含的字符的最大个数）。如果你想手工设置容量以避免不必要的缓存复制，可以使用

www.TopSage.com

reserve成员函数并且传递一个数值型参数给它以指明希望的缓存大小；尽管在可能的缓存的大小上有个最大数，并且你能通过调用缓存max_size函数得到它。在你的标准类库实现中，你可以使用这些函数来观察使用内存的增长情况。请看示例4-9是如何做的。

示例4-9 字符串长度和容量

```
#include <string>
#include <iostream>

using namespace std;

int main() {

   string s = "";
   string sr = "";

   sr.reserve(9000);

   cout << "s.length    = " << s.length()   << '\n';
   cout << "s.capacity  = " << s.capacity() << '\n';
   cout << "s.max_size  = " << s.max_size() << '\n';

   cout << "sr.length   = " << sr.length()   << '\n';
   cout << "sr.capacity = " << sr.capacity() << '\n';
   cout << "sr.max_size = " << sr.max_size() << '\n';

   for (int i = 0; i < 10000; ++i) {

 if (s.length() == s.capacity()) {
    cout << "s reached capacity of " << s.length()
    << ", growing...\n";
 }
 if (sr.length() == sr.capacity()) {
    cout << "sr reached capacity of " << sr.length()
    << ", growing...\n";
 }
 s += 'x';
 sr += 'x';
   }
}
```

在VC7.1的环境中，我的输出如下：

```
s.length    = 0
s.capacity  = 15
s.max_size  = 4294967294
sr.length   = 0
sr.capacity = 9007
sr.max_size = 4294967294
s reached capacity of 15, growing...
s reached capacity of 31, growing...
s reached capacity of 47, growing...
s reached capacity of 70, growing...
s reached capacity of 105, growing...
s reached capacity of 157, growing...
s reached capacity of 235, growing...
s reached capacity of 352, growing...
s reached capacity of 528, growing...
s reached capacity of 792, growing...
s reached capacity of 1188, growing...
s reached capacity of 1782, growing...
s reached capacity of 2673, growing...
s reached capacity of 4009, growing...
s reached capacity of 6013, growing...
```

www.TopSage.com

大家网 TopSage.com

```
sr reached capacity of 9007, growing...
s reached capacity of 9019, growing...
```

以上的情况是随着我不断地追加字符到字符串中，字符串所对应的缓存也不断地被填充。当缓存被填满时（如，length = capacity 时），一块新的更大的缓存被分配，并且原来字符串中的那些字符和新追加的字符就被复制到这个新分配的缓存中。上例字符串 s 开始时的容量是 15（不同编译器容量会有不同），然后，每次以大概一半的容量往上增长。

如果你预期到在你的字符串中会有重大的增长，或者你要使用大量的字符串并且不希望这些字符串有什么增长，你可以使用reserve成员函数指定容量的大小以使缓存被重新分配的次数达到最少。用标准类库的实现做个实验来观察它是如何处理字符串的增长情况是一个好的办法。

顺便说一下，当你想知道一个字符串是否为空，你可以不去看它的长度是否为零，而是调用 empty 成员函数。它是一个 const 成员函数，如果字符串的长度是零，它将返回真值。

4.5 翻转字符串

问题

如何翻转一个字符串？

解决方案

不使用临时字符串而使字符串位置翻转，可以使用 <algorithm> 头文件中的 reverse 函数模板：

```
std::reverse(s.begin(), s.end());
```

讨论

翻转工作非常简单：它修改你给定范围内的字符串以使范围内的字符的位置和原来的位置刚好相反。它花费线性时间。

当你想从后向前地把一个字符串复制到另一个字符串中时，你可以使用翻转迭代器，如下所述：

```
std::string s = "Los Angeles";
std::string rs;

rs.assign(s.rbegin(), s.rend());
```

rbegin和rend返回翻转迭代器。翻转迭代器看待一个序列是从后往前的。Rbegin返回的迭代器指向最后一个元素，而rend返回的迭代器指向第一个元素；这刚好和begin和 end 所做的相反。

www.TopSage.com

但是你需要首先考虑翻转字符串吗？使用rbegin和rend，任何在迭代器上进行操作的成员函数或算法都能在字符串的翻转版本上进行操作。如果你想对一个字符串进行查找操作，你可以使用rfind函数来做find函数能进行的任何操作，只是它从字符串的末端开始，然后从后往前进行查找。对于大字符串或大量的字符串，翻转操作的代价是昂贵的，所以应尽量避免使用。

4.6 分割字符串

问题

如何把含定界符的字符串分割成多个字符串。例如，你想去除掉"Name|Address|Phone"这个字符串中的定界符，从而把它分成三个分开的字符串，"Name"、"Address"和"Phone"。

解决方案

使用basic_string中的find成员函数依次找到每个定界符，然后使用substr函数把每个子字符串从原来的字符串中复制出来。你可以使用标准的序列来存储结果。在示例4-10中，使用vector：

示例4-10 分割含定界符的字符串

```
#include <string>
#include <vector>
#include <functional>
#include <iostream>

using namespace std;

void split(const string& s, char c,
 vector<string>& v) {
   string::size_type i = 0;
   string::size_type j = s.find(c);

   while (j != string::npos) {
 v.push_back(s.substr(i, j-i));
 i = ++j;
 j = s.find(c, j);

 if (j == string::npos)
    v.push_back(s.substr(i, s.length()));
   }
}

int main() {
   vector<string> v;
   string s = "Account Name|Address 1|Address 2|City";

   split(s, '|', v);

   for (int i = 0; i < v.size(); ++i) {
 cout << v[i] << '\n';
   }
}
```

www.TopSage.com

讨论

使得上面的函数模板例子能接受任意的字符是非常容易的，只要把字符的类型参数化并把引用从 string 改成 basic_string<T> 就可以：

```
template<typename T>
void split(const basic_string<T>& s, T c,
 vector<basic_string<T> >& v) {
   basic_string<T>::size_type i = 0;
   basic_string<T>::size_type j = s.find(c);

   while (j != basic_string<T>::npos) {
 v.push_back(s.substr(i, j-i));
 i = ++j;
 j = s.find(c, j);

 if (j == basic_string<T>::npos)
    v.push_back(s.substr(i, s.length(  )));
   }
}
```

它具有相同的逻辑。

注意： 请注意，尽管我只是在这个函数头部的最后一行的最右边两个括号之间加入一个额外的空格，你也不得不告诉编译器还没读到右移位操作符。

示例4-10使用了一个简单的算法来分割字符串。从前端开始，它寻找定界符c的第一个匹配位置，然后考虑它前面的所有字符和有意义的文本块起始处后面的所有字符。这个例子使用find成员函数从原始字符串特定索引处开始查找定界符的第一个匹配处，然后用 substr 函数把这个范围内的字符复制到另一个新的字符串中，并把这个新字符串放到 vector 中去。这和大部分脚本语言中分割函数的功能相同，事实上，对一个文本流进行特征符分割是一个特例，就像4.7节描述的那样。

一个普通的需求就是基于一个简单字符定界符来分割字符串，并且在Boost字符算法类库中也有相应的实现。而且非常容易使用，示例4-11就说明了如何使用Boost分割函数来分割字符串。

示例4-11 使用Boost来分割字符串

```
#include <iostream>
#include <string>
#include <list>
#include <boost/algorithm/string.hpp>

using namespace std;
using namespace boost;

int main() {

   string s = "one,two,three,four";
   list<string> results;

   split(results, s, is_any_of(","));  // Note this is boost::split
```

www.TopSage.com

```
   for (list<string>::const_iterator p = results.begin();
   p != results.end(); ++p) {
 cout << *p << endl;
   }
}
```

split是一个带有三个参数的函数模板。它的声明如下：

```
template<typename Seq,
    typename Coll,
    typename Pred>
Seq& split(Seq& s, Coll& c, Pred p,
   token_compress_mode_type e = token_compress_off);
```

类型Seq、Coll和Pred分别代表结果序列、输入集合和断言的类型，这个断言是用来判断是否是定界符的。这个序列就是C++标准定义中的序列，是用来包含输入集合的片段用的。因此，示例4-11我使用了list<string>，但你也可使用vector<string>或者别的类vector<string>的序列类型。集合参数是输入序列类型。集合是一个类似序列的非标准类型，具有更少限制（参考*www.boost.org*上的boost文档）。这个断言参数是一个一元函数对象或函数指针，用来返回一个布尔型的值来指明它的参数是否是定界符。它以f(*it)的形式调用序列中的每一个元素，f(*it)形式中的it是个迭代器，用来引用序列中的每个元素。

is_any_of是字符串算法库中一个方便的函数模板，使得你能容易地使用多个定界符。它构建了一个一元函数对象，返回真值的话表明你传给它的参数是这个集合当中的一员。换句话说：

```
bool b = is_any_of("abc")('a'); // b = true
```

它使得你在自己不亲自写函数对象时测试多个定界符非常的容易。

4.7 标记字符串

问题

如何使用一组定界符来把一个字符串分解成多个片段？

解决方案

使用basic_string的find_first_of和first_first_not_of成员函数来列举字符串并交替地定位下一个特征符和非特征符。示例4-12显示了一个实现此功能的简单的StringTokenizer类。

示例4-12 字符串特征化

```
#include <string>
#include <iostream>

using namespace std;

// String tokenizer class.
class StringTokenizer {
```

www.TopSage.com

```
public:

   StringTokenizer(const string& s, const char* delim = NULL) :
 str_(s), count_(-1), begin_(0), end_(0) {


 if (!delim)
    delim_ = " \f\n\r\t\v";  //default to whitespace
 else
    delim_ = delim;

 // Point to the first token
 begin_ = str_.find_first_not_of(delim_);
 end_ = str_.find_first_of(delim_, begin_);
   }

   size_t countTokens() {
if (count_ >= 0) // return if we've already counted
  return(count_);

string::size_type n = 0;
string::size_type i = 0;

for (;;) {
   // advance to the first token
   if ((i = str_.find_first_not_of(delim_, i)) == string::npos)
 break;
   // advance to the next delimiter
   i = str_.find_first_of(delim_, i+1);
   n++;
   if (i == string::npos)
break;
}
return (count_ = n);
   }
   bool hasMoreTokens() {return(begin_ != end_);}
   void nextToken(string& s) {
if (begin_ != string::npos && end_ != string::npos) {
   s = str_.substr(begin_, end_-begin_);
   begin_ = str_.find_first_not_of(delim_, end_);
   end_ = str_.find_first_of(delim_, begin_);
}
else if (begin_ != string::npos && end_ == string::npos)
{
   s = str_.substr(begin_, str_.length()-begin_);
   begin_ = str_.find_first_not_of(delim_, end_);
}

   }

private:
   StringTokenizer() {};
   string delim_;
   string str_;
   int count_;
   int begin_;
   int end_;
};

int main() {
   string s = " razzle dazzle giddyup ";
   string tmp;

   StringTokenizer st(s);

   cout << "there are " << st.countTokens() << " tokens.\n";
   while (st.hasMoreTokens()) {
 st.nextToken(tmp);
```

www.TopSage.com

大家网 TopSage.com

```
 cout << "token = " << tmp << '\n';
   }
}
```

讨论

用一个定义好的结构来分隔字符串（就像示例4-10那样）是比较好的，但并不是总是那么容易的。假设你不得不特征化一个字符串而不是简单地基于定界符把它分成几个片段。最通常的案例就是在忽视空白符的情况下特征化。示例4-12给出了一个简单的C++实现类StringTokenizer（像JAVA的类名），它接受定界符，且默认值就是空白符。

类StringTokenizer中最重要的行就是使用basic_string的find_first_of和find_first_not_of成员函数。在4.9节中我将描述它们是如何工作的、何时应该使用它们。示例4-10的输出如下：

```
There are 3 tokens.
token = razzle
token = dazzle
token = giddyup
```

StringTokenizer是示例4-10中分隔函数的更为灵活的形式。因此你能预先知道应从一个特征符移到下个特征符而不是一次分析整个输入字符串。同时，你还能计算出特征符的个数。

对StringTokenizer类，你仍然还有很多的改进。首先，为了简单起见，我仅仅只为字符串写了这个StringTokenizer类，换句话说，是为窄位字符串。如果你想使得同一个类既能为窄位字符串又能为宽位字符串工作，你就只好像我在前面几节中做的那样，把字符类型参数化。另外一个你可能想扩展StringTokenizer类的是使它与序列交互得更为友好，具有更好的扩展性。你可以一直实现自己的类，或者你也能使用已有的类。Boost类库中有一个叫做tokenizer的类，它就是用来实现此功能的。更详细的信息请参考 *www.boost.org*，也可参考4.24节。

4.8 连接字符串

问题

假设你有一个字符串序列，如示例4-10的输出那样，如何把它们连接成一个单一的长字符串，也可带分界符？

解决方案

循环地访问序列并把每个字符串都追加到输出字符串中。你可以用任何的标准序列当作输入；示例4-13使用了一个字符串的vector。

示例4-13 连接一个字符串序列

```
#include <string>
#include <vector>
#include <iostream>

using namespace std;

void join(const vector<string>& v, char c, string& s) {

   s.clear();

   for (vector<string>::const_iterator p = v.begin();
   p != v.end(); ++p) {
 s += *p;
 if (p != v.end() - 1)
   s += c;
   }
}

int main() {

   vector<string> v;
   vector<string> v2;
   string s;

   v.push_back(string("fee"));
   v.push_back(string("fi"));
   v.push_back(string("foe"));
   v.push_back(string("fum"));

   join(v, '/', s);

   cout << s << '\n';
}
```

讨论

示例4-13用到的一个技术和前面的例子稍有不同。请看下面：

```
for (vector<string>::const_iterator p = v.begin();
```

前面的字符串例子只是简单地使用迭代器，没有“const”部分，但这里你不能把它删掉，因为v是一个声明为const的对象。如果你有一个const的容器对象，你只能用const_iterator迭代器来访问它的每一个元素。这是因为一个简单的迭代器允许你改变它所引用的对象，但是，如果你的容器对象是const型的，则不允许你这样做。

我把v声明为const类型有两方面的原因。第一，我知道我将不会去改变它的内容，因此，如果我改变了，我希望编译器能告诉我一个错误。编译器在处理这类问题上比我做得更好，特别是由于一个敏感的语法或语义错误导致一个不合乎需要的赋值。第二，我想告诉这个函数的用户我不想对这个容器做任何的改变，const是实现此目的的最好的办法。现在，我只是不得不创建一个泛型函数版本，它能够处理多个字符类型。

就像在4.6节那样，用函数模板来实现join函数的泛型版本很容易。你所需要做的是使函数参数的字符类型参数化，如下所示：

```
template<typename T>
```

www.TopSage.com

```
void joing(const std::vector<std::basic_string<T> >& v, T c,
 std::basic_string<T>& s)
```

但vector可能并不是你唯一的输入。你可能还要把C风格的字符串也连接进来。C++的字符串比C风格的字符串好，因此，如果你不得不处理此种任务，把它们连接成C++的字符串更好。一旦你这样做了，你就可调用字符串的c_str成员函数来得到C风格的字符串版本；这个函数返回一个const型的指针，它指向一个以null结尾的字符数组。

示例4-14提供了一个一般意义上的连接版本，它把一系列的字符数组连接到一个字符串当中。由于这个新的一般意义上的版本中，字符类型是参数输入的，所以它可以处理窄位字符和宽位字符数组。

示例4-14 连接C风格的字符串

```
#include <string>
#include <iostream>

const static int MAGIC_NUMBER = 4;

template<typename T>
void join(T* arr[], size_t n, T c, std::basic_string<T>& s) {
   s.clear();

   for (int i = 0; i < n; ++i) {
 if (arr[i] != NULL)
    s += arr[i];
 if (i < n-1)
    s += c;
   }
}

int main() {
   std::wstring ws;

   wchar_t* arr[MAGIC_NUMBER];

   arr[0] = L"you";
   arr[1] = L"ate";
   arr[2] = L"my";
   arr[3] = L"breakfast";

   join(arr, MAGIC_NUMBER, L'/', ws);
}
```

4.9 在字符串中进行查找

问题

如何查找字符串？或许你想找到一个单一的字符，或者是另外一个字符串，或者一个无序字符集的序列。而且，由于你自己的原因，你不得不以某种特殊的方式查找，例如，第一个匹配的或最后一个匹配的，或者相对于特定索引的第一个或最后一个匹配处。

解决方案

使用basic_string的find成员函数。几乎所有的以单词“find”开始的函数，并且它们的名字非常好地说明了它们的功能。示例4-15就说明了find成员函数是如何工作的。

www.TopSage.com

示例4-15 查询字符串

```
#include <string>
#include <iostream>

int main() {
   std::string s = "Charles Darwin";

   std::cout << s.find("ar") << '\n';              // Search from the
                                                   // beginning
   std::cout << s.rfind("ar") << '\n';             // Search from the end

   std::cout << s.find_first_of("swi")             // Find the first of
   << '\n';                                        // any of these chars

   std::cout << s.find_first_not_of("Charles")     // Find the first
   << '\n';                                        // that's not in this
                                                   // set

   std::cout << s.find_last_of("abg") << '\n';     // Find the first of
                                                   // any of these chars
                                                   // starting from the
                                                   // end

   std::cout << s.find_last_not_of("aDinrw")       // Find the first
   << '\n';                                        // that's not in this
                                                   // set, starting from
                                                   // the end
}
```

每个find成员函数都将在“讨论”部分详细地讲解。

讨论

有六个查找成员函数用来查找字符串，它们中每一个都提供了四个重载函数。允许为basic_string或charT*（一种字符类型）参数进行重载。每一个重载函数都有一个basic_string::size_type参数pos，它用来让你能够指明查找开始处的索引，并且其中一个重载函数带有size_type参数n，它是用来让你能够只用这个集合中的最初的n个字符进行匹配。

给每个这样的成员函数都给出详细的说明是困难的，因此表4-2给出了每个函数及其参数的一个快速索引。

表4-2 字符串查找成员函数

成员函数	描述
size_type find (const basic_string& str, size_type pos = 0) const; size_type find (const charT* s, size_type pos, size_type n) const; size_type find (const charT* s, size_type pos = 0) const;	返回从参数pos指定索引开始的第一个匹配字符或第一个匹配子字符串，如果指定了参数n，则将查找与这个目标字符串的最初n个字符中相匹配的字符或子字符串

www.TopSage.com

表 4-2 字符串查找成员函数（续）

成员函数	描述
size_type find (charT c, size_type pos = 0) const;	
size_type rfind (...)	从后向前查找字符串中的第一个匹配的字符或子字符串。换句话说，这个函数和 find 函数具有相同的功能，只是从字符串的末端开始查找
size_type find_first_of (...)	查找在basic_string或字符指针提供的字符集合中第一个匹配的任意的字符。如果指定了参数 n 的话，则只是用这个集合的最初 n 个字符进行匹配
size_type find_last_of (...)	查找在basic_string或字符指针提供的字符集合中最后一个匹配的任意的字符。如果指定了参数 n 的话，则只是用这个集合的最初 n 个字符进行匹配
size_type find_first_not_of (...)	查找第一个不在basic_string或字符指针提供的字符集合中出现的字符。如果指定了参数 n 的话，则只是查找不在这个集合的最初 n 个字符中的字符
size_type find_last_not_of (...)	查找最后一个不在 basic_string 或字符指针提供的字符集合中出现的字符。如果指定了参数n的话，则只是查找不在这个集合的最初 n 个字符中的字符

所有的这些成员函数都以basic_string<T>::size_type类型的值返回所查找字符的索引。如果查找失败，它返回 basic_string<T>::npos，它是一个特殊的值（常常是 -1），表明查找失败。尽管它常常是 -1，但是为平台可移植性考虑你还是应该用npos来测试是否查找失败；这样才能使你的意图更为明显，通过跟npos比较而不是跟一个别的数比较，能让人明白你正在确认查找是否失败。

通过这些各种各样的查找算法，你应该能找到你正在查找的字符或字符串，如果不能，你也可以用你自己的查找算法进行查找。如果 basic_string 不能提供你所需要的，在使

www.TopSage.com

用你自己的算法前，还是应该仔细看看<algorithm>。这个标准的算法通过迭代器常常也是函数对象作用在序列上。方便的是，basic_string提供了迭代器，使得在元素对象上移动非常容易，因此，把字符串上的迭代器安插到标准算法上没有多大的意义。假设你想从一行字符串中，找到第一处两个紧挨着的相同字符的出现位置。你可以使用adjacent_find函数模板来从一个字符串中查找两个相同的紧邻两个元素("adjacent"意思是在一个迭代器中，它们的位置是不同的，例如，*iter == * (iter + 1))。

```
std::string s = "There was a group named Kiss in the 70s";

std::string::iterator p =
   std::adjacent_find(s.begin(), s.end());
```

这个结果是一个指向第一处相同的相邻元素的迭代器。

如果你为了操作字符串而不得不自己写算法，为了利用已有的成员函数，请不要像C语言使用[]操作符那样使用basic_string来获取每一个元素。每一个这样的find函数都带一个size_type类型的参数以指定查找位置开始处的索引。重复使用find函数，你就能预知你的字符串构造。示例4-16说明如何计算字符串中不同字符的个数。

示例4-16 计算不同字符的个数

```
#include <string>
#include <iostream>

template<typename T>
int countUnique(const std::basic_string<T>& s) {
   using std::basic_string;

   basic_string<T> chars;

   for (typename basic_string<T>::const_iterator p = s.begin();
   p != s.end(); ++p) {
if (chars.find(*p) == basic_string<T>::npos)
   chars += *p;
   }
   return(chars.length());
}

int main() {
   std::string s = "Abracadabra";

   std::cout << countUnique(s) << '\n';
}
```

find函数是十分方便的。当你想从字符串中查找时请首先考虑使用它。

4.10 查找第n个子字符串

问题

假设你有两个字符串：源字符串和模式字符串，如何在源字符串中找到第*n*个与模式相匹配的子字符串？

www.TopSage.com

解决方案

使用find成员函数来定位你正在查找子串的连续多个实例。示例4-17包含了一个简单的nthSubstr函数。

示例4-17 定位一个子串的第n个实例

```
#include <string>
#include <iostream>

using namespace std;

int nthSubstr(int n, const string& s,
    const string& p) {
   string::size_type i = s.find(p); // Find the first occurrence

   int j;
   for (j = 1; j < n && i != string::npos; ++j)
 i = s.find(p, i+1); // Find the next occurrence

   if (j == n)
return(i);
   else
return(-1);
}

int main() {
   string s = "the wind, the sea, the sky, the trees";
   string p = "the";

   cout << nthSubstr(1, s, p) << '\n';
   cout << nthSubstr(2, s, p) << '\n';
   cout << nthSubstr(5, s, p) << '\n';
}
```

讨论

示例4-17中的nthSubstr还有很多可以改进之处。首先，你可以把它改变成一个函数模板代替原来的普通函数使得它更为一般。第二，你可以增加一个参数说明子串可以或不可以自己相交迭。这里的交迭，我的意思是字符串的开始部分与字符串末端部分相匹配。就像字符串“abracadabra”中，最后的四个字符和开始的四个字符相同。示例4-18说明这种实现。

示例4-18 nthSubstr的一个改进版本

```
#include <string>
#include <iostream>

using namespace std;

template<typename T>
int nthSubstrg(int n, const basic_string<T>& s,
const basic_string<T>& p,
bool repeats = false) {
   string::size_type i = s.find(p);
   string::size_type adv = (repeats) ? 1 : p.length();

   int j;
   for (j = 1; j < n && i != basic_string<T>::npos; ++j)
 i = s.find(p, i+adv);
```

www.TopSage.com

```
   if (j == n)
return(i);
   else
return(-1);
}

int main() {
   string s = "AGATGCCATATATATACGATATCCTTA";
   string p = "ATAT";

   cout << p << " as non-repeating occurs at "
   << nthSubstrg(3, s, p) << '\n';
   cout << p << " as repeating occurs at "
   << nthSubstrg(3, s, p, true) << '\n';
}
```

示例4-18的输出如下：

```
ATAT as non-repeating occurs at 18
ATAT as repeating occurs at 11
```

请参考4.9节。

4.11 从字符串中删除一个子字符串

问题

如何从字符串中删除掉一个子串？

解决方案

使用basic_string的find、erase和length成员函数：

```
std::string t = "Banana Republic";
std::string s = "nana";

std::string::size_type i = t.find(s);

if (i != std::string::npos)
   t.erase(i, s.length());
```

这将从find函数第一次发现的子串的开始处的索引处删除s.length()个元素。

讨论

找到一个子字符串并删除它有很多种变种。例如，可能想删除一个子字符串所有示例而不是仅仅一个；或者最后一个；或者第七个。每次的算法都差不多：找到你想删除的模式开始处的索引，然后在这个索引处调用erase函数删除接下来的n个字符，其中n是指模式字符串的长度。用于字符串查找的不同的find成员函数，请参考4.9节。

如果你使你的子字符串删除函数类属化，你就可以在任意类型字符的字符串上使用它。示例4-19给出了一个在字符串上删除掉所有跟模式字符串相匹配的子字符串的删除函数的泛型版本。

www.TopSage.com

示例4-19 删除字符串的所有子串（泛型版本）

```
#include <string>
#include <iostream>

using namespace std;

template<typename T>
void removeSubstrs(basic_string<T>& s,
    const basic_string<T>& p) {
   basic_string<T>::size_type n = p.length();

   for (basic_string<T>::size_type i = s.find(p);
   i != basic_string<T>::npos;
   i = s.find(p))
 s.erase(i, n);
}

int main() {
   string s = "One fish, two fish, red fish, blue fish";
   string p = "fish";

   removeSubstrs(s, p);

   cout << s << '\n';
}
```

这里basic_string的成员函数erase起到了重要的作用。在<string>中，它被重载了三次。示例4-19中使用的版本接受删除元素开始处的索引和要删除的元素的个数。另一个版本则接受元素开始处和结尾处的迭代器作为参数，还有一个版本是带一个单一的迭代器并删除迭代器所指的元素。为了得到最佳的性能，当你想删除多个相邻的元素时，推荐使用前两个版本而不是为每个你想删除的元素重复调用s.erase(iter)。换句话说，当你想修改字符串或序列的值时，推荐使用成员函数操作在一个范围上而不是操作在一个特定的元素上。这样做的话，你就能避免为序列中的每个元素都额外地调用erase函数，并且，这样的话你也能让字符串的实现能更加聪明地管理它自己的数据。

4.12 将字符串转换为小写或大写

问题

如何把一个字符串转换成小写或大写？

解决方案

使用<cctype>头文件中的toupper和tolower函数来把字符转换成大写或小写。示例4-20表明了如何来使用这些函数。“讨论”部分将给出一个替代方案。

示例4-20 转换字符串大小写

```
#include <iostream>
#include <string>
#include <cctype>
#include <cwctype>
#include <stdexcept>
```

www.TopSage.com

```
using namespace std;

void toUpper(basic_string<char>& s) {
   for (basic_string<char>::iterator p = s.begin();
   p != s.end(); ++p) {
 *p = toupper(*p); // toupper is for char
   }
}

void toUpper(basic_string<wchar_t>& s) {
   for (basic_string<wchar_t>::iterator p = s.begin();
   p != s.end(); ++p) {
 *p = towupper(*p); // towupper is for wchar_t
   }
}

void toLower(basic_string<char>& s) {
   for (basic_string<char>::iterator p = s.begin();
   p != s.end(); ++p) {
 *p = tolower(*p);
   }
}

void toLower(basic_string<wchar_t>& s) {
   for (basic_string<wchar_t>::iterator p = s.begin();
   p != s.end(); ++p) {
 *p = towlower(*p);
   }
}

int main() {

   string s = "shazam";
   wstring ws = L"wham";

   toUpper(s);
   toUpper(ws);

   cout << "s =  " << s << endl;
   wcout << "ws = " << ws << endl;

   toLower(s);
   toLower(ws);

   cout << "s =  " << s << endl;
   wcout << "ws = " << ws << endl;
}
```

输出如下：

```
s =  SHAZAM
ws = WHAM
s =  shazam
ws = wham
```

讨论

你可能会以为标准的string类有一个成员函数来把所有的字符转换成大写或小写，但是，事实上，不是这样。如果你想把一个字符串中的所有字符转换成大写或小写，在一定程度上，你不得不自己实现。

不必惊奇，不只一种办法可用来转换字符串的大小写（这里的字符串是指一个字符序列，可能是窄字符也可能是宽字符）。最简单的办法就是调用这四个转换函数：toupper、

www.TopSage.com

towupper、tolower和 towlower中的一种，每一种的第一个形式是窄字符版本，第二个（带w）是宽字符形式。

这些函数中的每一个都是使用当前的全局大小写转换的语言环境规则进行字符的大小写转换的。大写或小写都依赖于当前使用的语言环境中的字符；一些字符就没有对应的大写或小写字符，在这种情况下，上面的这些函数将返回与传给它的相同的字符。关于语言环境更为详细的请参考第13章。C++用来处理不同语言环境的技术是复杂的，这里我不可能详细加以介绍。

事实上字符的大小写转换的实现还是容易的。下面就是示例4-20中的toUpper函数：

```
void toUpper(basic_string<char>& s) {
   for (basic_string<char>::iterator p = s.begin();
   p != s.end(); ++p) {
 *p = toupper(*p);
   }
}
```

黑体部分的那行是真正地转为大写字符的工作。宽字符的版本也基本上相同：

```
void toUpper(basic_string<wchar_t>& s) {
   for (basic_string<wchar_t>::iterator p = s.begin();
   p != s.end(); ++p) {
 *p = towupper(*p);
   }
}
```

我用不同的字符类型重载了toUpper函数，因为没有泛型的toupper函数来转换字符的大小写（除非你用<locale>头文件中的facet）。以上两个简单的函数将完成大写字符的转换工作。

还有另外一种方法来实现这种工作，采用这种方法的出发点是你需要显式地语言环境locale。下面的toUpper和toLower版本的函数转换字符串的大小写，而不管字符的类型，只需要有一个命名的locale（默认是当前的locale）来对这个语言环境和字符类型提供大小写转换工作。

```
template<typename C>
void toUpper2(basic_string<C>& s, const locale& loc = locale()) {
   typename basic_string<C>::iterator p;
   for (p = s.begin(); p != s.end(); ++p) {
 *p = use_facet<ctype<C> >(loc).toupper(*p);
   }
}

template<typename C>
void toLower2(basic_string<C>& s, const locale& loc = locale( ) {
   typename basic_string<C>::iterator p;
   for (p = s.begin(); p != s.end(); ++p) {
 *p = use_facet<ctype<C> >(loc).tolower(*p);
   }
}
```

黑体部分的代码行完成实际的工作。从功能上讲，它们与示例4-20中使用的大写和小写

www.TopSage.com

转换函数相同，只是它们使用的是<locale>中的国际化技术。关于locale、facet和国际化的更为详细的讨论请参考第13章。

4.13 大小写不敏感的字符串比较

问题

有两个字符串，在不考虑字符大小写的情况下，如何知道它们是否相同？例如，“cat”与“dog”不相同，但与“Cat”相同，同样与“cat”、“CAT”和“caT”都相同。

解决方案

使用equal标准算法（定义在<algorithm>）来比较两个字符串，为了比较大写字符串你需要提供你自己的使用<cctype>文件中的toupper函数（或<cwctype>文件中宽字符的towupper函数）的比较函数。示例4-21提供了一个通用方案。它说明STL的使用和它的灵活性；详细的解释请看如下的讨论。

示例4-21 大小写不敏感的字符串的比较

```
#include <string>
#include <iostream>
#include <algorithm>
#include <cctype>
#include <cwctype>

using namespace std;

inline bool caseInsCharCompareN(char a, char b) {
   return(toupper(a) == toupper(b));
}

inline bool caseInsCharCompareW(wchar_t a, wchar_t b) {
   return(towupper(a) == towupper(b));
}

bool caseInsCompare(const string& s1, const string& s2) {
   return((s1.size() == s2.size()) &&
  equal(s1.begin(), s1.end(), s2.begin(),
caseInsCharCompareN));
}

bool caseInsCompare(const wstring& s1, const wstring& s2) {
   return((s1.size() == s2.size()) &&
  equal(s1.begin(), s1.end(), s2.begin(),
caseInsCharCompareW));
}

int main() {
   string s1 = "In the BEGINNING...";
   string s2 = "In the beginning...";
   wstring ws1 = L"The END";
   wstring ws2 = L"the endd";

   if (caseInsCompare(s1, s2))
  cout << "Equal!\n";
```

www.TopSage.com

```
    if (caseInsCompare(ws1, ws2))
  cout << "Equal!\n";
}
```

讨论

大小写不敏感的字符串比较的最主要的部分就是字符串中对应字符是否相等的判断，因此让我们来首先讨论这部分。在这节中，由于我使用equal标准算法，但我又想使用我的特殊的比较标准，所以我不得不创建一个独立的函数来处理我的特殊比较。

示例4-21中第9～15行定义了字符比较函数caseInsCharCompareN和caseInsCharCompareW。它们分别使用toupper和towupper来把每个字符转换成大写字符并返回它们是否相等的指示。

一旦我完成了我的比较函数，我就可以使用标准的算法把我的比较函数应用到任意的字符序列中。第17～25行使用equal函数定义了caseInsCompare函数来完成此项工作。为我关注的每个字符类型定义了两个重载函数。当然它们的功能相同，但每个都会为它自己的字符类型调用相应的字符比较函数。例如，我重载了两个普通函数，但你可以使用模板达到同样的效果。请参考“我是否应使用模板？”讨论部分。

Equal为了判断是否相等比较两个序列。有两个版本：一个是使用操作符==，另一个是使用你提供的任何二元断言（例如，带两个参数并返回一个布尔值）函数对象。示例4-21中，caseInsCharCompareN和caseInsCharCompareW都是二元断言函数。

但光做这些还不够，你还需要比较它们的长度。请看equal函数的声明：

```
template<typename InputIterator1, typename InputIterator2,
    typename BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1,
 InputIterator2 first2, BinaryPredicate pred);
```

假设从first1到last1中有*n*个元素，也就是说，第一个序列有*n*个元素。如果这两个序列的前*n*个元素相等，equal函数将返回真。也就意味着，当两个序列的前*n*个元素相等时，尽管第二个序列有多于*n*个元素，equal函数也返回真。如果加上长度比较的话，就能避免这种错误的发生。

你不需要在函数中封装这种逻辑。你的代码或用户的代码还是可以直接调用这个算法，但无论你什么时候想按大小写不敏感地比较字符串，如下这样写更容易记住也更清晰：

```
if (caseInsCompare(s1, s2)) {
// they are equal, do something
```

比如下这样写：

```
if ((s1.size() == s2.size()) &&
    std::equal(s1.begin(), s1.end(), s2.begin(),
  caseInsCharCompare<char>)) {
// they are equal, do something
```

www.TopSage.com

4.14 大小写不敏感的字符串查找

问题

如何在不管大小写的情况下从一个字符串中查找一个子串？

解决方案

你可以使用标准的定义在<algorithm>中的transform和search算法，以及你自己的特殊字符比较函数，如在上一节那样。示例4-22描述了是如何实现的。

示例4-22 大小写不敏感的字符串查找

```
#include <string>
#include <iostream>
#include <algorithm>
#include <iterator>
#include <cctype>

using namespace std;

inline bool caseInsCharCompSingle(char a, char b) {
   return(toupper(a) == b);
}

string::const_iterator caseInsFind(string& s, const string& p) {
   string tmp;

   transform(p.begin(), p.end(),        // Make the pattern
   back_inserter(tmp),                  // upper-case
   toupper);

   return(search(s.begin(), s.end(),    // Return the iter-
  tmp.begin(), tmp.end(),               // ator returned by
  caseInsCharCompSingle));              // search
}

int main( ) {
   string s = "row, row, row, your boat";
   string p = "YOUR";
   string::const_iterator it = caseInsFind(s, p);

   if (it != s.end()) {
 cout << "Found it!\n";
   }
}
```

通过返回一个指向模式字符串在目标字符串中起始位置处的元素的迭代器，你就能很容易地和其他标准算法进行比较，因为它们都接受迭代器作为参数。

讨论

示例4-22说明了一个和标准算法一起工作时的常用操作模式。创建一个做实际工作的函数，然后把它作为函数对象安插到最适合的算法中去。charInsCharCompSingle是实际上完成工作的函数，不像示例4-21那样，这个比较函数只把第一个参数变成大写字符。这是因为稍后的caseInsFind函数中，为了避免多次把模式字符串变成大写字符，在使用它之前，我就把它变成了大写字符。

www.TopSage.com

如果这个比较函数可以的话，就使用transform和search标准算法来完成这两项工作。使用transform把整个模式字符串（不是目标字符串）转成大写。然后，使用search函数和比较函数在目标字符串中定位子串。

记住，标准算法操作在序列而不是字符串上。它们总的来说是主要但不唯一操作在标准容器上，但它们不假设容器的内容。所有的标准算法都需要你提供一个比较函数（如果没有，它们就用默认的），你的比较函数只是比较两个元素，并返回一个布尔值来表明这两个元素是否相同。

示例4-22中有一点看上去奇怪的是caseInsCompare返回了一个const_iterator迭代器，如下所示：

```
string::const_iterator caseInsFind(const string& s,
const string& p)
```

如果你想修改这个返回的迭代器所指向的元素会怎样？这是个合理需求。它是const类型的原因是因为传给caseInsFind函数的字符串是const类型的，因此，你不能用一个非const类型的迭代器索引一个const字符串。如果你用一个普通的迭代器，你就能修改字符串，从它的参数中删除掉const修饰符并把这个函数的声明改成返回一个string::iterator，就可以达到这个目的。

4.15 在文本文件中进行制表符与空格的相互转换

问题

有一个包含制表符和空格的文本文件，如何把制表符和空格相互转换？例如，你可能想用三个空格替换所有的制表符，或者相反，有可能想用一个制表符替换一定数量的空格。

解决方案

不管是用空格替换制表符还是用制表符替换空格，都可使用定义在<fstream>中的ifstream和ofstream类。在下面的第一个（更简单）的例子中，从输入流中读数据，一次一个字符，检查它，如果是制表符，就写一定数量的空格到输出流中去。示例4-23描述了是如何实现的。

示例4-23 用空格替换制表符

```
#include <iostream>
#include <fstream>
#include <cstdlib>

using namespace std;

int main(int argc, char** argv) {

   if (argc < 3)
 return(EXIT_FAILURE);
```

www.TopSage.com

大家网 TopSage.com

```
   ifstream in(argv[1]);
   ofstream out(argv[2]);

   if (!in || !out)
return(EXIT_FAILURE);

   char c;
   while (in.get(c)) {
 if (c == '\t')
    out << "   "; // 3 spaces
 else
    out << c;
   }

   out.close();

   if (out)
 return(EXIT_SUCCESS);
   else
 return(EXIT_FAILURE);
}
```

相反，你如果想用制表符替换空格，请看示例4-24。它包含spacesToTabs函数，它从输入流中读数据，一次一个字符，寻找连续的三个空格。当它找到了在一行中有三个空格时，就写一个制表符到输出流中去。对于别的字符，或者比三个更少的空格，从输入流读的任何字符都写到输出流中去。

示例4-24 用制表符代替空格

```
#include <iostream>
#include <istream>
#include <ostream>
#include <fstream>
#include <cstdlib>

using namespace std;

void spacesToTabs(istream& in, ostream& out, int spaceLimit) {

   int consecSpaces = 0;
   char c;

   while (in.get(c)) {
 if (c != ' ') {
    if (consecSpaces > 0) {
  for (int i = 0; i < consecSpaces; i++) {
out.put(' ');
  }
  consecSpaces = 0;
    }
    out.put(c);
 } else {
    if (++consecSpaces == spaceLimit) {
  out.put('\t');
  consecSpaces = 0;
    }
 }
   }
}

int main(int argc, char** argv) {

   if (argc < 3)
 return(EXIT_FAILURE);
```

www.TopSage.com

```
   ifstream in(argv[1]);
   ofstream out(argv[2]);

   if (!in || !out)
 return(EXIT_FAILURE);

   spacesToTabs(in, out, 3);

   out.close();

   if (out)
 return(EXIT_SUCCESS);
   else
 return(EXIT_FAILURE);
}
```

讨论

两个方案机制相同；只是算法上稍有不同。使用get从输入流中读字符，使用put向输出流中写字符。把转换逻辑放在调用这两个函数之间。

你大概注意到了在示例4-24中的main函数中，我把in和out分别声明成ifstream和ofstream，并且spacesToTabs函数的参数事实上也是istream和ostream。我这样做的原因主要是想让spacesToTabs函数在任何流(任何继承自basic_istream或basic_ostream的流)上都能工作，并不仅仅是文件流。例如，你可能想把你的文本重新格式化成字符流(<sstream>中的istringstream和ostringstream)。这种情况下，只用和下面一样：

```
istringstream istr;
ostringstream ostr;

// fill up istr with text...

spacesToTabs(istr, ostr);
```

和字符串相比，流其实是一个类模板，其参数是流操作的字符的类型。例如，ifstream是basic_ifstream<char>使用typedef重新定义的，而wifstream是basic_ifstream<wchar_t>使用typedef定义的。因此，你如果希望例子4-23和4-24中的spacesToTabs函数能在任何字符的流上工作，你可以用类模板来代替typedef：

```
template<typename T>
void spacesToTabs(std::basic_istream<T>& in,
   std::basic_ostream<T>& out,
   int spaceLimit) { //...
```

4.16 文本文件中的换行

问题

在一个文件中，如何在特定数目的字符后进行自动换行？例如，你想在72个字符后给文本换行，你就在文件中每72个字符后插入一个换行符。如果文件包含可读的文本，在自动换行时，还应避免把可读的单词拆开。

www.TopSage.com

解决方案

实现一个函数，它用istream::get(char)从输入流中读字符，用ostream::put(char)向输出流中写字符，实现书签的工作。示例4-25描述了一个简单的例子，实现了文本自动换行的同时又不拆开可读字符。

示例4-25 文本自动换行

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string>
#include <cctype>
#include <functional>

using namespace std;

void textWrap(istream& in, ostream& out, size_t width) {

   string tmp;
   char cur = '\0';
   char last = '\0';
   size_t i = 0;

   while (in.get(cur)) {
 if (++i == width) {
    ltrimws(tmp);   // ltrim as in Recipe
    out << '\n' << tmp;  // 4.1
    i = tmp.length();
    tmp.clear();
 } else if (isspace(cur) &&   // This is the end of
  !isspace(last)) { // a word
    out << tmp;
    tmp.clear();
 }
 tmp += cur;
 last = cur;
   }
}

int main(int argc, char** argv) {
   if (argc < 3)
 return(EXIT_FAILURE);

   int w = 72;
   ifstream in(argv[1]);
   ofstream out(argv[2]);

   if (!in || !out)
return(EXIT_FAILURE);

   if (argc == 4)
w = atoi(argv[3]);

   textWrap(in, out, w);

   out.close();

   if (out)
 return(EXIT_SUCCESS);
   else
 return(EXIT_FAILURE);
}
```

www.TopSage.com

讨论

textWrap从输入流中一次读一个字符。每个字符都追加到一个临时字符串tmp中，直到它达到了单词的末尾或行的最大宽度。如果它达到了单词的末尾，但没达到行的最大宽度，这个临时字符串就输出到输出流中去。否则，如果达到了行的最大宽度，一个新行就写到输出流中去，同时这个临时字符串的前面的空白符去掉，并且把这个临时字符串输出到输出流中去。按照这种方式，textWrap将会写足够多的字符到输出流中去而不超过行的最大宽度。它不是拆开一个单词，而是把达到行的最大宽度的单词输出到下一行。

示例4-25使用的流和4.15节中的基本相同。关于这些流的详细信息请参考4.15节。

4.17 计算文本文件中的字符数、字数和行数

问题

如何计算文本文件中的字符、字数和行数以及其他类型的文本元素的统计。

解决方案

使用输入流读字符，一次一个，随着你读到字符、字和行标志，增加相应的统计。示例4-26包含了函数countStuff，它就是用来完成这种工作的。

示例4-26 统计文本文件的元素

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cctype>

using namespace std;

void countStuff(istream& in,
 int& chars,
 int& words,
 int& lines) {

   char cur = '\0';
   char last = '\0';
   chars = words = lines = 0;

   while (in.get(cur)) {
 if (cur == '\n' ||
(cur == '\f' && last == '\r'))
    lines++;
 else
   chars++;
 if (!std::isalnum(cur) &&   // This is the end of a
std::isalnum(last))          // word
    words++;
 last = cur;
   }
   if (chars > 0) {          // Adjust word and line
 if (std::isalnum(last))     // counts for special
```

www.TopSage.com

```
    words++;                    // case
 lines++;
   }
}

int main(int argc, char** argv) {

   if (argc < 2)
 return(EXIT_FAILURE);

   ifstream in(argv[1]);

   if (!in)
 exit(EXIT_FAILURE);

   int c, w, l;

   countStuff(in, c, w, l);
1
   cout << "chars: " << c << '\n';
   cout << "words: " << w << '\n';
   cout << "lines: " << l << '\n';
}
```

讨论

这个算法是好理解的。字符数比较容易统计：每次从输入流中得到一个字符就增加相应的统计数。行稍微复杂点，因为行结束标志依赖于操作系统。不过，它常常是一个新行符（\n）或回车换行符（\r\l）。通过跟踪当前和随后的字符，找到这种字符还是容易的。单词则说难就难，说易就易，因为它依赖于你对单词的定义。

就示例4-26来说，我认为一个连续的文字数字字符块就是一个字。当我检查输入流中的每个字符时，如果我碰到非文字数字字符，我就检查它的前一个字符，看它是否是文字数字字符。如果是，则一个字就结束了，我就给字数加1。我是用<cctype>文件中的isalnum函数来判断一个字符是否是文字数字字符的。但并不只是这个函数有这个功能，你可以用类似的函数来测试字符的不同性质。表4-3中列出了很多这种函数。对宽字符，使用这些在“is”后面带“w”的函数就可以，例如iswspace。宽字符的函数声明在<cwctype>文件中。

表4-3 <cctype>和<cwctype>中的字符测试函数

函数	描述
isalpha iswalpha	希腊字母：a-z, A-Z（大写或小写）
isupper iswupper	大写希腊字母：A-Z
islower iswlower	小写希腊字母：a-z
isdigit iswdigit	数字：0-9

www.TopSage.com

大家网 TopSage.com

表 4-3 <cctype> 和 <cwctype> 中的字符测试函数（续）

函数	描述
isxdigit iswxdigit	十六进制数字：0-9, a-f, A-F
isspace iswspace	空白符：' ', \n, \t, \v, \r, \l
iscntrl iswcntrl	控制符：ASCII 0-31 和 127
ispunct iswpunct	不属于以上各组的标点符号
isalnum iswalnum	isalpha 或 isdigit 为真
isprint iswprint	可打印的 ASCII 字符
isgraph iswgraph	isalpha 或 isdigit 或 ispunct 为真

当所有的字符都读进来后，流就到达末端，还是有一点需要调整。首先，循环只是计算了换行符，严格地说，这不是行数。因此，它总是比事实上的行数少 1。为了消除这个问题，当文件中有多于 0 个字符时，给行数加 1。第二，如果流是以字母或数字结尾的话，最后一个单词的测试就永远不会发生，因为不可能测试接下来的字符。为了解决这种问题，我测试流中最后一个字符是不是字母和数字（也是在文件中有多于 0 个字符的情况下），如果是的话，我就给它的字数加 1。

示例 4-26 中流的使用和 4.14 节和 4.15 节中的基本相同，但更简单，因为只是检查整个文件而不做任何修改。

请参考 4.14 节和 4.15 节。

4.18 计算文本文件中每个字的出现次数

问题

如何计算文本文件中每个字出现的次数？

解决方案

使用定义在 <string> 中的 >> 操作符，从输入文件中读一个连续的文本块，并使用定义在 <map> 文件中的 map 数据结构，存储每个单词和它在文件中出现的频率。示例 4-27 描述了这种方案。

www.TopSage.com

示例4-27 计算单词的频率

```
#include <iostream>
#include <fstream>
#include <map>
#include <string>

typedef std::map<std::string, int> StrIntMap;

void countWords(std::istream& in, StrIntMap& words) {

   std::string s;

   while (in >> s) {
 ++words[s];
   }
}

int main(int argc, char** argv) {

   if (argc < 2)
 return(EXIT_FAILURE);

   std::ifstream in(argv[1]);

   if (!in)
 exit(EXIT_FAILURE);

   StrIntMap w;
   countWords(in, w);

   for (StrIntMap::iterator p = w.begin();
   p != w.end(); ++p) {
 std::cout << p->first << " occurred "
 << p->second << " times.\n";
   }
}
```

讨论

示例4-27非常简单，但还有可改进的地方。大部分的这种解决方案都和数据结构map有关，让我们先来谈谈它。

如果你还不熟悉map的话，应该熟悉一下它。map是一个容器类模板，是C++标准模板类库STL的一部分。它根据随同std::less或你提供的定制比较函数按序存储键－值对，你可以根据你的需要来存储任何类型的键和值，在类型上，它们没有特别的限制。在这个例子中，我们存储字符串和整数值。

在第6行中我使用typedef来重新定义它使代码更清晰：

```
typedef map<string, int> StrIntMap;
```

因此，StrIntMap是一个map结构，存储字符串/整数对。字符串是一个唯一的单词，这也是我为什么把它当成关键字的原因，它是从文件中读进来的，相对应的整数值是它出现的次数。余下应该做的就是每次从文件中按一次读一个单词的方式把单词读进来，如果这个单词在map中没有，则把它加到map中去，并把整数值设为1；如果它在map中存在，则把它对应的整数值增加1。

www.TopSage.com

这就是 countWords 的工作。主要的逻辑如下：

```
while (in >> s) {
   ++words[s];
}
```

操作符>>从它的左边操作数(流)读一块连续的非空文本块并把它放到右边的操作数(字符串）中去。一旦读完一个字，我所做的就是更新 map 结构中的统计数，如下所示：

```
++words[s];
```

为了取到一个给定关键字对应的值，map 定义了操作符[](它返回的是值的索引)，因此增加它，也就是增加某个特定的关键字对应的值。但是或许你还有点疑问，如果这个关键字不在map中会怎样？难道我能够增加一个不存在的索引所对应的值，并和数组一样导致程序崩溃吗？不会的，map 的[]操作和其他 STL 的容器或正常的 C 风格的数组有些不同。

map 中，操作符[]会做两件事：如果这个关键字不存在的话，它会利用值类型的默认的构造函数构造出一个值对象并把这个新的键－值对加到 map 中去，如果这个关键字存在的话，则不做任何修改。在这两种情况下，与这个关键字对应的值的索引被这个操作符返回，尽管在第一种情况下，这个值是用默认的构造函数构造的。这是一个便利的功能(你如果知道的话)，因为使得用户的代码在插入一个关键字时还要提前验证它是否已经存在。

现在，请看第 32 和 33 行。这个迭代器引用了两个成员变量：first 和 second，它们是什么呢？ map给你玩了个魔术，它引用了另一个类模板来存储你的键－值对：定义在 <utility>（已经包含在 <map> 中）中的pair类模板。如果你正在列举存储在 map 中的键－值对，你将指向一个 pair 对象。pair 对象是简单的，存储在 pair 对象中的第一个项目就是第一个成员，第二个项目当然就是第二个成员。

从一个输入流中，我用操作符>>读一个连续的文本块，这一点和别的例子有点不同。这里我只是示范一下可以这样用，你如果需要用的话，应该根据你文本文件中字的定义来相应地修改它的行为。例如，考虑示例 4-27 输出的摘要：

```
with occurred 5 times.
work occurred 3 times.
workers occurred 3 times.
workers. occurred 1 times.
years occurred 2 times.
years. occurred 1 times.
```

请注意，单词后面的句号也被当成是单词的一部分。大部分情况下，你想改变字的定义以使它只是字母或数字，就像我在 4.17 节中所做的那样使用 <cctype> 和 <cwctype> 中的一些字符测试函数。

请参考 4.17 节和表 4-3。

www.TopSage.com

4.19 给文本文件添加空白

问题

给定一个文本文件，如何给它添加空白？换句话说，你想给每行的两边增加一些字符，因此每一行就具有同样的宽度。

解决方案

示例4-28描述了如何使用流、字符串和getline函数模板给一个文件添加空白。

示例4-28 给文本文件添加空白

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>

using namespace std;
const static char PAD_CHAR = '.';

// addMargins takes two streams and two numbers. The streams are for
// input and output. The first of the two numbers represents the
// left margin width (i.e., the number of spaces to insert at the
// beginning of every line in the file). The second number represents
// the total line width to pad to.
void addMargins(istream& in, ostream& out,
 int left, int right) {

   string tmp;

   while (!in.eof()) {
 getline(in, tmp, '\n');                          // getline is defined
                                                  // in <string>
 tmp.insert(tmp.begin(), left, PAD_CHAR);
 rpad(tmp, right, PAD_CHAR);                      // rpad from Recipe
                                                  // 4.2
 out << tmp << '\n';
   }
}

int main(int argc, char** argv) {

   if (argc < 3)
 return(EXIT_FAILURE);

   ifstream in(argv[1]);
   ofstream out(argv[2]);

   if (!in || !out)
 return(EXIT_FAILURE);

   int left = 8;
   int right = 72;

   if (argc == 5) {
 left = atoi(argv[3]);
 right = atoi(argv[4]);
   }

   addMargins(in, out, left, right);
```

www.TopSage.com

```
    out.close();

    if (out)
  return(EXIT_SUCCESS);
    else
  return(EXIT_FAILURE);
}
```

这个例子给读进来的文本格式做了一些假设，因此一定要读一下接下来的讨论部分。

讨论

addMargins函数假设你的输入看上去和下面一样：

```
The data is still inconclusive. But the weakness
in job creation and the apparent weakness in
high-paying jobs may be opposite sides of a coin.
Companies still seem cautious, relying on
temporary workers and anxious about rising health
care costs associated with full-time workers.
```

这个文本是在50个字符处就自动换行（参考4.16节），而且是左对齐的（参考4.20节）。addMargins函数还假设你的输出看上去与下面的相同，它使用了句号来代替空格以使你能看出哪些是添加的字符：

```
........The data is still inconclusive. But the weakness.............
........in job creation and the apparent weakness in.................
........high-paying jobs may be opposite sides of a coin.............
........Companies still seem cautious, relying on....................
........temporary workers and anxious about rising health............
........care costs associated with full-time workers..................
```

默认的是，左边的添加的是8个字符，并且行的总长是72个字符。当然，如果你知道你的输入总是左对齐或右对齐的话，你就能简单地在每行添加你能预见个数的字符。无论如何，逻辑还是很简单明了的。这节大部分技术（流、给字符串添加字符）都已讨论过了，这些就不再讨论了。没有出现过的函数是getline。

如果你想一次读一行文本，或者更精确，你想把文本读进来直到碰到一个特定的定界符，使用定义在<string>中的getline函数模板，就像我在示例4-28中做的那样：

```
getline(in, tmp, '\n');
```

以上的getline函数从输入流中读字符并把字符追加到tmp中去，直到碰到定界符'\n'为止，这个定界符不加到tmp中去。basic_istream中有个同名的函数，但它的行为不一样。它把它的输出存到一个字符缓存中而不是字符串中。这个例子中，我想利用字符串的成员函数，并不想把一行字符读进冗余的字符缓存中，而是把它复制到一个字符串中，因此我使用字符串版本的getline函数。

请参考4.16节和4.20节。

www.TopSage.com

大家网 TopSage.com

4.20 调整文本文件

问题

如何使你的文本右对齐或左对齐？

解决方案

使用流和标准流的格式标志右和左，它们都是定义在<ios>中的ios_base的一部分。示例4-29描述了是如何实现的。

示例4-29 调整文本的版面

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>

using namespace std;

int main(int argc, char** argv) {

   if (argc < 3)
 return(EXIT_FAILURE);

   ifstream in(argv[1]);
   ofstream out(argv[2]);

   int w = 72;
   if (argc == 4)
 w = atoi(argv[3]);

   string tmp;
   out.setf(ios_base::right);    // Tell the stream to
                                 // right-justify
   while (!in.eof()) {
 out.width(w);                   // Reset width after
 getline(in, tmp, '\n');         // each write
 out << tmp << '\n';
   }
   out.close();
}
```

这个例子带有三个参数：一个输入文件、一个输出文件和右对齐的宽度。你可以使用如下的输入文件：

```
With automatic download of Microsoft's (Nasdaq:
MSFT) enormous SP2 security patch to the Windows
XP operating system set to begin, the industry
still waits to understand its ramifications. Home
users that have their preferences set to receive
operating-system updates as they are made
available by Microsoft may be surprised to learn
that some of the software they already run on
their systems could be disabled by SP2 or may run
very differently.
```

使它的输出如下：

```
 With automatic download of Microsoft's (Nasdaq:
MSFT) enormous SP2 security patch to the Windows
```

www.TopSage.com

```
    XP operating system set to begin, the industry
  still waits to understand its ramifications. Home
   users that have their preferences set to receive
     operating-system updates as they are made
   available by Microsoft may be surprised to learn
 that some of the software they already run on
  their systems could be disabled by SP2 or may run
    very differently.
```

这第二个文本例子右对齐到50个字符。

讨论

ios_base类模板中有很多标志可用来格式化从流读进来的和写入流的数字和文本数据。控制文本对齐的是right和left。它们都是ios_base中的静态常量成员，都是fmtflags（定义实现的）类型。所有的这些都定义在<ios>中。

可使用ios_base::setf来设置格式标志。你设置的标志和流中已存在的标志是“或”关系。例如，下面这行代码打开了右对齐的功能：

```
out.setf(std::ios_base::right);
```

但右对齐如果没有设置右边页面的空白宽度的话就没有意义。为了设置这个宽度，可使用ios_base::width，如下所示：

```
out.width(w);
```

这将把输出的字段对齐的宽度设置成你传入的值大小，也就是说，当你进行右对齐时，在你的字符串的右边的起始处将会增加你设置的宽度那么多的空格字符。注意我是在循环内设置宽度的，而设置右对齐的标志则是在循环的前面。我这样做的原因是每次向流输出一次后，宽度会自动重新设为0，而格式标志则不会，所以我可以初始化一次格式标志然后一直用下去。

简洁和负责任总是好的，所以当你用完你设置的格式标志时，你应该做的就是清掉掉它们。

通常，你正在使用的流并不只属于你，特别是你正在写一个通用的库或API时。例如，如果你写一个奇特的记录函数，它带有一个输出流和一个字符串，你修改这个字符串，设置格式标志，并把这个字符串写到流中去，你就可能带来负面的影响。当用户调用你的记录函数时，它的流能有它自己的格式标志，你的记录函数就有可能改变它。办法是在你的函数中把原来的复制下来，当你的函数运行完时，恢复它。

例如，下面就是一个负责任的错误记录函数：

```
using namespace std;

void logError(ostream& out, const string& s) {

   string tmp(s);
   tmp.insert(0, "ERROR: ");
```

www.TopSage.com

```
    ios_base::fmtflags flgs =    // setf returns the
  out.setf(ios_base::left);      // flags that were
                                 // already there
    out.width(72);
    out << tmp << '\n';

    out.flags(flgs);             // reset to original
  }
```

flags函数和setf功能基本相同，但你传给它的标志和流中已有的标志不是“或”的关系，而是替代它们。因此，当你调用flags函数并把原来的标志传给它时，就清除了你设置的标志。

4.21 将文本文件中的多个空格压缩成一个

问题

有一个带有不同长度空白符的文本文件，如何把文件中连续的多个空格减少成一个？

解决方案

使用定义在<string>中的操作符>>函数模板，从一个流中把一块连续的非空白字符读到一个字符串中。然后使用它的对等操作符<<，把每块文本写到输出流中去，并在每块的末尾追加一个空格符。示例4-30给出了运用这种技术的简单例子。

示例4-30 把空白符压缩成单个空格

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main(int argc, char** argv) {

   if (argc < 3)
 return(EXIT_FAILURE);

   ifstream in(argv[1]);
   ofstream out(argv[2]);

   if (!in || !out)
 return(EXIT_FAILURE);

   string tmp;

   in >> tmp;            // Grab the first word
   out << tmp;           // Dump it to the output stream

   while (in >> tmp) {   // operator>> ignores whitespace, so all I have
 out << ' ';             // to do is add a space and each chunk of non-
 out << tmp;             // whitespace
   }

   out.close();
}
```

www.TopSage.com

讨论

如果你能利用流和字符串的话，这将是一件简单的事情。即使你不得不实现这个例子的变种的话，你仍可以保留生成的新行。你如果想增加新行，可以使用4.16节中的方案把它们插到正当的位置。

请参考4.15节和4.16节。

4.22 自动改正文本

问题

有一个代表文本字段或文档的类，且有文本增加到其中，如何自动修改拼写错的单词，如同Microsoft Word“自动更正”的功能那样？

解决方案

使用map（定义在<map>文件中）、字符串和各种标准类库的功能，你就能用比较少的代码完成此功能。示例4-31描述了是如何实现的。

示例4-31 自动校对文本

```
#include <iostream>
#include <string>
#include <cctype>
#include <map>

using namespace std;

typedef map<string, string> StrStrMap;

// Class for holding text fields
class TextAutoField {

public:
   TextAutoField(StrStrMap* const p) : pDict_(p) {}
  ~TextAutoField() {}

   void append(char c);
   void getText(string& s) {s = buf_;}

private:
   TextAutoField();
   string buf_;
   StrStrMap* const pDict_;
};

// Append with autocorrect
void TextAutoField::append(char c) {

   if ((isspace(c) || ispunct(c)) &&     // Only do the auto-
  buf_.length() > 0 &&                   // correct when ws or
  !isspace(buf_[buf_.length() - 1])) {   // punct is entered

 string::size_type i = buf_.find_last_of(" \f\n\r\t\v");

 i = (i == string::npos) ? 0 : ++i;
```

www.TopSage.com

```
 string tmp = buf_.substr(i, buf_.length() - i);
 StrStrMap::const_iterator p = pDict_->find(tmp);

 if (p != pDict_->end()) {              // Found it, so erase
    buf_.erase(i, buf_.length() - i);   // and replace
    buf_ += p->second;
 }
   }
   buf_ += c;
}

int main() {

   // Set up the map
   StrStrMap dict;
   TextAutoField txt(&dict);

   dict["taht"] = "that";
   dict["right"] = "wrong";
   dict["bug"] = "feature";

   string tmp = "He's right, taht's a bug.";
   cout << "Original: " << tmp << '\n';
   for (string::iterator p = tmp.begin();
   p != tmp.end(); ++p) {
 txt.append(*p);
   }

   txt.getText(tmp);

   cout << "Corrected version is: " << tmp << '\n';
}
```

示例4-31的输出如下：

```
Original: He's right, taht's a bug.
Corrected version is: He's wrong, that's a feature.
```

讨论

当你不得不跟踪与字符串相关的东西时，使用map和字符串是比较方便的。TextAutoField是一个用字符串来存储数据的缓存。比较有意思的是它的append方法，它是用来侦听空白符合标点符号的，当它碰到它们时，就做相应的处理。

为了实现自动更正的功能，你需要做两件事。首先，你需要一个排过序的字典，它用来存储一些单词的通常的错误拼写和它对应的正确拼写。map存储键－值对，且键和值都可以是任何类型，因此map是一个理想的选择。在上面的示例4-31中，用了typedef来重新定义一个字符串对的map：

```
typedef map<string, string> StrStrMap;
```

更多的关于map的解释请参考4.18节。TextAutoField存储了一个指向map的指针，因为大部分情况下，你希望这个单一的字典能被所有字段使用。

假设用户的代码向map中加入了一些有意义的东西，append方法不得不周期性地在这个map中做查找工作。示例4-31中，append方法等待着空格和标点符号的到来然后运行它的功能。你可以用isspace函数测试一个字符是否是空格，或者你也可以用

www.TopSage.com

大家网 TopSage.com

ispunct 函数测试一个字符是否是标点符号，这两个函数都是定义在 <cctype> 文件中，并都是给窄字符使用的（请看表 4-3）。

如果你不熟悉如何使用 STL 容器的迭代器和 find 方法，这段查找功能的代码还是需要一些解释的。字符串 tmp 包含了追加到 TextAutoField 中的最后一块文本。在字典中查看它是否是通常的拼写错误，如下所示：

```
StrStrMap::iterator p = pDict_->find(tmp);

if (p != pDict_->end()) {
```

其中最重要的一点是 map::find 返回一个迭代器，如果在 map 中找到了这个字符串，这个迭代器就指向包含这个匹配成功的字符串键－值对；如果没找到，这个返回的迭代器就会指向 map 的尾部，就像 map::end 返回的那样（所有的 STL 容器都是这样实现 find 函数的）。如果在 map 中找到了这个单词，就从缓存中把它删除并用拼写正确单词代替它：

```
buf_.erase(i, buf_.length() - i);
buf_ += p->second;
```

再追加这个处理过的字符（空格或标点符号），这样你就自动更正完了。

请参考 4.17 节、4.18 节和表 4-3。

4.23 读取由逗号分隔开的文本文件

问题

如何读取一个由逗号和换行符（或其他的定界符）分隔的文本文件？记录被一个字符分开，而记录中的字段被另一个字符分开。例如，下面是一个逗号分隔开的雇员信息文本文件：

```
Smith, Bill, 5/1/2002, Active
Stanford, John, 4/5/1999, Inactive
```

这种文件常常是电子制表软件、数据库和其他文件格式导出的数据集的中间存储结果。

解决方案

示例 4-32 描述了如何实现。你如果调用 getline（定义在 <string> 中的一个函数模板）读一块连续的文本字符串，你就能使用 4.6 节中的分割函数来解析文本并把它放到一个数据结构当中，这个例子中是 vector。

示例 4-32 把一个有定界符的文件读入

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
```

www.TopSage.com

```
using namespace std;

void split(const string& s, char c,
 vector<string>& v) {
   int i = 0;
   int j = s.find(c);

   while (j >= 0) {
 v.push_back(s.substr(i, j-i));
 i = ++j;
 j = s.find(c, j);

 if (j < 0) {
    v.push_back(s.substr(i, s.length()));
 }
   }
}

void loadCSV(istream& in, vector<vector<string>*>& data) {

   vector<string>* p = NULL;
   string tmp;

   while (!in.eof()) {
 getline(in, tmp, '\n');       // Grab the next line

 p = new vector<string>();
 split(tmp, ',', *p);          // Use split from
                               // Recipe 4.7
 data.push_back(p);

 cout << tmp << '\n';
 tmp.clear();
   }
}

int main(int argc, char** argv) {

   if (argc < 2)
 return(EXIT_FAILURE);

   ifstream in(argv[1]);

   if (!in)
 return(EXIT_FAILURE);

   vector<vector<string>*> data;

   loadCSV(in, data);

   // Go do something useful with the data...

   for (vector<vector<string>*>::iterator p = data.begin();
   p != data.end(); ++p) {
 delete *p;                    // Be sure to de-
   }                           // reference p!
}
```

讨论

示例4-32已经没有什么可以讨论的了。getline在4.19节中讨论了，而vector在4.3节中讨论了。唯一值得讨论的是内存的分配。

loadCSV给每行新读进来的数据创建了一个新的vector并把这个vector存到另一个

www.TopSage.com

指向vector的vector中。因为每个这样的vector都是在堆上分配的，得有人把它释放掉，这个人就是你（不是vector的实现者）。

vector本身并不知道它包含的是值呢，还是指向值的指针，或者别的什么东西。它所知道的只是当它被删除时，它需要给它包含的每个元素调用析构函数。如果vector中存储的是对象，这很好；因为对象能被正确地释放。但如果包含的是指针，只是指针被删除，但并不删除指针指向的对象。

有两种办法可保证内存被释放掉。首先，你可以和示例4-32那样做，第二，和下面一样手工处理：

```
for (vector<vector<string>*>::iterator p = data.begin();
p != data.end(); ++p) {
   delete *p;
}
```

或者你可以使用引用计数指针，如Boost项目中的smart_ptr，它将会是即将到来的C++0X标准的一部分。但这样做是重要的，因此我推荐你读一下smart_ptr介绍和工作原理。更多关于Boost的信息，请参考*www.boost.org*。

4.24 使用正则表达式来分割字符串

问题

如何把一个字符串分割成多个特征符？你需要非常复杂的查找或提供比4.7节提供的更灵活的方案。例如，你可能需要多于一个字符的特征符或具有不同形式的特征符。在代码当中经常需要这样做，且经常给你的类或函数的用户造成混淆。

解决方案

使用Boost库中的regex类模板。regex使得在字符串和文本数据中使用正则表达式成为可能。示例4-33说明如何使用regex来分割字符串。

示例4-33 使用Boost正则表达式

```
#include <iostream>
#include <string>
#include <boost/regex.hpp>

int main() {

   std::string s = "who,lives:in-a,pineapple    under the sea?";

   boost::regex re(",|:|-|\\s+");       // Create the reg exp
   boost::sregex_token_iterator         // Create an iterator using a
p(s.begin(), s.end(), re, -1);          // sequence and that reg exp
   boost::sregex_token_iterator end;    // Create an end-of-reg-exp
                                        // marker
   while (p != end)
 std::cout << *p++ << '\n';
}
```

www.TopSage.com

讨论

示例4-33说明如何使用regex来列举和正则表达式相匹配的字符串。下面这行建立了这个正则表达式：

```
boost::regex re(",|:|-|\\s+");
```

它的意思是和这个正则表达式相匹配的是逗号，或者冒号，或者减号，或者是一个或多个空格。竖线表示两边的分割符是逻辑或的关系。下面两行创建了迭代器：

```
boost::sregex_token_iterator
   p(s.begin(), s.end(), re, -1);
boost::sregex_token_iterator end;
```

迭代器p是通过这个正则表达式和输入字符串来创建的。一旦它被成功创建，你就可以把p当成是标准类库中序列上的迭代器。不带参数的sregex_token_iterator对象是一个特殊的对象，它代表一个正则表达式特征符序列的末尾，通过和它进行比较你就能知道是否已经到达了特征符的末端。

www.TopSage.com

第5章

日期与时间

5.0 概述

日期和时间是两个非常广阔和复杂的主题。就因为这个事实，C++标准类库中没有提供合适的日期类型。C++给日期和时间的操作继承了C的结构和函数，并实现了一对考虑到本地化的日期/时间的输入输出函数。在由Jeff Garland开发的Boost date_time类库中，你能找到一些解脱，这个库是C++在日期和时间两个主题上最全面的和最具有扩展性的日期和时间的类库。在以后的很多章节中，我都将使用它。在C++社区中有一个期望就是未来的C++标准类库在日期和时间上的扩展将基于Boost date_time类库。

Boost date_time类库在操作日期和时间上包含两个单独的系统：一个是使用阳历日历来操作时间，另一个是使用阳历日历来操作日期。本章将讨论这两个主题。

关于日期和时间更多的信息，特别是读和写的，请参考第13章。

5.1 获得当前日期与时间

问题

如何从用户的计算机中获得当前的日期和时间，并把它当作是当地时间或同等的世界时间？

解决方案

调用来自<ctime>头文件中的时间函数，以0作为参数转给它。其结果是一个time_t

www.TopSage.com

大家网 TopSage.com

阳历日历和闰年

当前在西方世界中阳历日历是使用最广的日历。阳历日历原先是设计用来修正公历日历的缺点的。采用阳历日历的过程开始于1582年。

公历日历规定每个第四年就是一个闰年，但是每个第一百年不是闰年。阳历日历又带来了一个新的例外即每个第四百年也是一个闰年。

闰年是设计用来补偿地球绕着太阳转与日子长度之间产生的不同步。换句话说，一个太阳年的长度被日子的长度除，产生的结果是个无理数。结果就会变成如果日历不做调准，我们就将会有季节性的变动，这样在每个新年春分和冬至（区分季节）都将变得更不一致。

结构的值。你可以使用gmtime函数把这个time_t值转换成一个代表当前世界时间（又叫格林尼治标准时间）的tm结构；或者你可以用localtime函数把这个time_t值转换成一个代表当地时间的tm结构。示例5-1中的程序就是获取当前的日期、时间，然后把它转换成当地时间输出。接下来这个程序则是把它转换成世界时间然后输出。

示例5-1 获取当地时间和世界时间

```
#include <iostream>
#include <ctime>
#include <cstdlib>

using namespace std;

int main()
{
  // Current date/time based on current system
  time_t now = time(0);

  // Convert now to tm struct for local timezone
  tm* localtm = localtime(&now);
  cout << "The local date and time is: " << asctime(localtm) << endl;

  // Convert now to tm struct for UTC
  tm* gmtm = gmtime(&now);
  if (gmtm != NULL) {
cout << "The UTC date and time is: " << asctime(gmtm) << endl;
  }
  else {
    cerr << "Failed to get the UTC date and time" << endl;
    return EXIT_FAILURE;
  }
}
```

讨论

time函数返回一个time_t类型的值，它表示一个实现定义的算法类型，用来代表一个时间段（也叫做时间间隔）。使用time_t能代表的最大的时间间隔是2 147 483 648秒，或者大概是68年。

调用time(0)返回一个time_t值，代表的时间间隔是从一个实现定义好的基础时间（通常是1970年1月1日的零时零分零秒）到当前时间这段时间间隔。

2038年将会是一个问题

由于一个time_t仅仅被要求代表一个68年的时间间隔，而很多实现把1970年当作基础年来计算当前时间，那么许多流行的C++实现在2038年以后就不可能来计算日期和时间。也就意味着如果程序员提前采取足够的预防措施的话，在2038年很多软件都会有问题。

一个更有用的代表当前日期和时间的做法是调用localtime或gmtime函数把它转换成一个tm结构。示例5-2中说明了一个tm结构中有一个整型字段。

示例5-2 tm结构的组成部分

```
struct tm {
  int tm_sec;   // seconds of minutes from 0 to 61 (60 and 61 are leap seconds)
  int tm_min;   // minutes of hour from 0 to 59
  int tm_hour;  // hours of day from 0 to 24
  int tm_mday;  // day of month from 0 to 23
  int tm_mon;   // month of year from 0 to 11
  int tm_year;  // year since 1900
  int tm_wday;  // days since sunday
  int tm_yday;  // days since January 1st
  int tm_isdst; // hours of daylight savings time
}
```

当使用gmtime函数时，一定要验证它的返回值。如果这台正在运行代码的计算机没有事先定义好当地时区的话，gmtime函数将不能计算出它的世界时间，并返回0。你如果把0传给asctime函数，它的行为是不可预知的。

localtime、gmtime和asctime函数都返回一个指向静态分配的对象的指针。对于类库来说，这是非常有效的，但是这也意味着随后的调用将改变这些对象的值。示例5-3中的代码描述了会产生惊奇的影响。

示例5-3 使用asctime函数的缺陷

```
void f() {
  char* x = asctime(localtime(time(0)));
  wait_for_15_seconds(); // do some long processing task
  asctime(localtime(time(0)));
  cout << x << endl; // prints out the current time, not fifteen seconds ago.
}
```

5.2 将日期/时间格式化为一个字符串

问题

如何把一个日期或时间格式化成一个字符串？

www.TopSage.com

解决方案

你可以如示例5-4那样使用定义在<locale>头文件中的time_put模板类。

示例5-4 格式化一个日期/时间字符串

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cstring>
#include <string>
#include <stdexcept>
#include <iterator>
#include <sstream>

using namespace std;

ostream& formatDateTime(ostream& out, const tm& t, const char* fmt) {
  const time_put<char>& dateWriter =
  use_facet<time_put<char> >(out.getloc());
  int n = strlen(fmt);
  if (dateWriter.put(out, out, ' ', &t, fmt, fmt + n).failed()) {
    throw runtime_error("failure to format date time");
  }
  return out;
}

string dateTimeToString(const tm& t, const char* format) {
  stringstream s;
  formatDateTime(s, t, format);
  return s.str();
}

tm now() {
  time_t now = time(0);
  return *localtime(&now);
}

int main()
{
  try {
    string s = dateTimeToString(now(), "%A %B, %d %Y %I:%M%p");
    cout << s << endl;
    s = dateTimeToString(now(), "%Y-%m-%d %H:%M:%S");
    cout << s << endl;
  }
  catch(...) {
    cerr << "failed to format date time" << endl;
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

示例5-4中程序的输出依赖于你的本地语言环境设置，与下面的字符类似：

```
Sunday July, 24 2005 05:48PM
2005-07-24 17:48:11
```

讨论

time_put成员函数与C语言中printf函数一样使用一个字符串来指定输出的格式。格式化后的字符被输出到一个缓存中，除非在字符前有一个特殊字符%。跟在%后的字

www.TopSage.com

符是用来指定输出格式的，它们具有特定的含义，表5-1说明了这些字符的含义。这些格式字符也支持一定的变形，如用一个整型数来指定字段的宽度，如%4B。

表5-1 日期/时间格式的指定符

格式指定符	描述
a	周日的缩写名（如Mon）
A	周日的完整名（如Monday）
b	月份的缩写名（如Dec）
B	月份的完整名（如May）
c	完全的日期和时间
d	月份中的天数（01~31）
H	24小时制中的小时数（00~23）
I	12小时制中的小时数（01~12）
j	一年中的天数（001~366）
m	月份数（01~12）
M	分钟数（00~59）
p	上午/下午的指定（AM/PM）
S	秒数，包括最多两个闰秒
U	周数（00~53），第一周从第一个星期天开始
w	每周中的天数（0~6），星期天是0
W	周数（00~53），第一周从第一个星期一开始
x	采用MM/DD/YY格式的日期
X	24小时制中HH/MM/SS格式的时间
y	当前世纪中的年数（00~99）
Y	年
Z	时区的缩写，如果系统不知道时区的话，就是空

以后几节中要讨论的Boost的date_time类库没有time_put函数提供的这些格式化日期时间的功能。为了方便，示例5-5给出了几个能把Boost日期/时间类转换成tm结构的函数，所以在tm结构的基础上你就可以使用time_put函数了。

示例5-5 把Boost的日期/时间类转换成tm结构

```
using boost::gregorian;
using boost::posix_time;

void dateToTmAux(const date& src, tm& dest) {
  dest.tm_mday = src.day();
```

www.TopSage.com

```
  dest.tm_year = src.year() - 1900;
  dest.tm_mon = src.month() - 1;
}

void ptimeToTmAux(const ptime& src, tm& dest) {
  dest.tm_sec = src.seconds();
  dest.tm_min = src.minutes();
  dest.tm_hour = src.hours();
  dateToTmAux(src.date(), dest);
}

tm ptimeToTm(const ptime& t) {
  tm ret = tm();
  ptimeToTmAux(t, ret);
  return ret;
}
```

请参考13.3节。

5.3 日期与时间运算

问题

如何知道在两个日期/时间点之间有多长时间?

解决方案

如果这两个日期/时间点都落在1970年到2038年之间，你就可使用定义在<ctime>头文件中的difftime函数和time_t类型。示例5-6描述了如何计算两个日期之间有多少天。

示例5-6 使用time_t类型的日期和时间的算法

```
#include <ctime>
#include <iostream>
#include <cstdlib>

using namespace std;

time_t dateToTimeT(int month, int day, int year) {
  // january 5, 2000 is passed as (1, 5, 2000)
  tm tmp = tm();
  tmp.tm_mday = day;
  tmp.tm_mon = month - 1;
  tmp.tm_year = year - 1900;
  return mktime(&tmp);
}

time_t badTime() {
  return time_t(-1);
}

time_t now() {
  return time(0);
}

int main() {
  time_t date1 = dateToTimeT(1,1,2000);
  time_t date2 = dateToTimeT(1,1,2001);
```

www.TopSage.com

大家网 TopSage.com

```
  if ((date1 == badTime()) || (date2 == badTime())) {
    cerr << "unable to create a time_t struct" << endl;
    return EXIT_FAILURE;
  }
  double sec = difftime(date2, date1);
  long days = static_cast<long>(sec / (60 * 60 * 24));
  cout << "the number of days between Jan 1, 2000, and Jan 1, 2001, is ";
  cout << days << endl;
  return EXIT_SUCCESS;
}
```

示例 5-6 程序输出如下：

```
the number of days between Jan 1, 2000, and Jan 1, 2001, is 366
```

请注意，2000 是闰年，因为它能被 100 整除，同时又能被 400 整除，因此它有 366 天。

讨论

Time_t 是一个实现上定义好的算法类型。也就意味着它是一个整型或浮点型，因此它就支持基本的算术操作。你可以执行加、减、乘、除等操作。为了计算两个 time_t 值之间的秒数，你需要使用 difftime 函数。不要假设 time_t 自己能计算秒数，即使是这样的。C++ 的许多实现可能在不久的将来把它改成计算成几分之一秒（这也是为什么 difftime 函数返回一个 double 类型的值的一个原因）。

如果你不能满足 time_t 类型的限制条件的话，你可以使用 Boost 的 date_time 类库中各种类来计算时间间隔。示例 5-7 说明如何使用 Boost 类来计算 20 世纪和 21 世纪之间的天数。

示例 5-7 日期和时间算法函数 date_duration

```
#include <iostream>
#include <boost/date_time/gregorian/gregorian.hpp>

using namespace std;
using namespace boost::gregorian;

int main()
{
  date_duration dd = date(2000, 1, 1) - date(1900, 1, 1);
  cout << "The twentieth century had " << dd.days() << " days" << endl;
  dd = date(2100, 1, 1) - date(2000, 1, 1);
  cout << "The twenty-first century will have " << dd.days() << " days" <<
endl;
}
```

示例 5-7 中程序的输出如下：

```
The twentieth century had 36524 days
The twenty-first century will have 36525 days
```

5.4 在时区之间转换

问题

如何把一个时区的当前时间转换成另一个时区的当前时间？

www.TopSage.com

解决方案

为了在不同的时区间转换时间，可以使用Boost的date_time类库中的时区转换函数。示例5-8说明如何把纽约时间转换成亚利桑那州图森市的时间。

示例5-8 在不同的时区之间转换时间

```
#include <iostream>
#include <boost/date_time/gregorian/gregorian.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>
#include <boost/date_time/local_time_adjustor.hpp>

using namespace std;
using namespace boost::gregorian;
using namespace boost::date_time;
using namespace boost::posix_time;

typedef local_adjustor<ptime, -5, us_dst> EasternTZ;
typedef local_adjustor<ptime, -7, no_dst> ArizonaTZ;

ptime NYtoAZ(ptime nytime) {
  ptime utctime = EasternTZ::local_to_utc(nytime);
  return ArizonaTZ::utc_to_local(utctime);
}

int main()
{
   // May 1st 2004,
   boost::gregorian::date thedate(2004, 6, 1);
   ptime nytime(thedate, hours(19)); // 7 pm
   ptime aztime = NYtoAZ(nytime);
   cout << "On May 1st, 2004, when it was " << nytime.time_of_day().hours();
   cout << ":00 in New York, it was " << aztime.time_of_day().hours();
   cout << ":00 in Arizona " << endl;
}
```

示例5-8中的程序输出如下：

```
On May 1st, 2004, when it was 19:00 in New York, it was 16:00 in Arizona
```

讨论

示例5-8中的时区转换经历了两个过程。第一，把要转换的时间转换成世界时间，然后再把这个世界时间转换成第二个时区的时间。请注意，Boost的date_time类库中用local_adjustor模板类来代表时区。每个类型都有转换函数，用来把给定的时间转换成对应的世界时间（local_to_utc函数），并且把世界时间转换成对应的给定时区的时间（utc_to_local函数）。

5.5 确定一年中的某天

问题

如何得到某一年中的某天？例如，1月1日是每年的第1天；2月5日是每年的第36天等。但是因为一些年份在2月28日后有闰年的天数，所以一个给定日期在每一年中可能具有不同天数的顺序号。

www.TopSage.com

解决方案

这个问题的解决方案同时需要好几个问题的解决方案。首先，你需要知道每个月份有多少天，同时，你又得知道这一年是否是闰年。示例5-9提供了这个问题的解决办法。

示例5-9 确定一年中的某天

```
#include <iostream>

using namespace std;

enum MonthEnum  {
  jan = 0, feb = 1, mar = 2, apr = 3, may = 4, jun = 5,
  jul = 6, aug = 7, sep = 8, oct = 9, nov = 10, dec = 11
};

bool isLeapYear(int y) {
  return (y % 4 == 0) && ((y % 100 != 0) || (y % 400 == 0));
}

const int arrayDaysInMonth[] = {
  31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
};

int n;
int arrayFirstOfMonth[] = {
  n = 0,
  n += arrayDaysInMonth[jan],
  n += arrayDaysInMonth[feb],
  n += arrayDaysInMonth[mar],
  n += arrayDaysInMonth[apr],
  n += arrayDaysInMonth[may],
  n += arrayDaysInMonth[jun],
  n += arrayDaysInMonth[jul],
  n += arrayDaysInMonth[aug],
  n += arrayDaysInMonth[sep],
  n += arrayDaysInMonth[::oct],
  n += arrayDaysInMonth[nov]
};

int daysInMonth(MonthEnum month, int year) {
  if (month == feb) {
    return isLeapYear(year) ? 29 : 28;
  }
  else {
    return arrayDaysInMonth[month];
  }
}

int firstOfMonth(MonthEnum month, int year) {
  return arrayFirstOfMonth[month] + isLeapYear(year);
}

int dayOfYear(MonthEnum month, int monthDay, int year) {
  return firstOfMonth(month, year) + monthDay - 1;
}

int main() {
  cout << "July 1, 1971, was the " << dayOfYear(jul, 1, 1971);
  cout << " day of the year" << endl;
}
```

示例5-9程序的输出如下：

```
July 1, 1971, was the 181 day of the year
```

www.TopSage.com

讨论

示例5-9中的代码是非常简单的，使用了一序列的函数来处理日期和闰年。请注意我放弃了前一节中我称之为“文档和祈祷”的方法。意思是我不再使用索引数字来代表月份而是使用枚举来代表月份。当把一个月份作为参数传给函数时能大大减少程序员出错的可能性。

示例5-9中的闰年计算是根据现代罗马日历的。每个第四年就是一个闰年，那些能被100整除但不能被400整除的年份不是（例如，1896年是闰年，1900年不是，2000年是，2004年也是，2100年不是）。

5.6 定义受限数据类型

问题

你需要验证数字类型来定义合法值具有有限范围的数值，如一天的小时数或一个小时的秒数。

解决方案

当你计算日期和时间时，常常希望这些值是在一个有限范围内的合法的整数（如，一分钟内的秒数是0~59，一天的小时数是0~23，一年的天数是0~365之间的整数值）。你更希望通过重载赋值操作符来使得这些值能够自动地验证，而不是每次把这些值传给某个函数时都验证一下。由于这种类型的值有很多，实现一个简单的类，使得这个类能处理不同数值范围的验证工作是一种非常好的解决方案。示例5-10描述了一个名为ConstrainedValue模板类的实现，它使得定义有限范围的整型值和其他的受限值类型变得容易。

示例5-10 constrained_value.hpp

```
#ifndef CONSTRAINED_VALUE_HPP
#define CONSTRAINED_VALUE_HPP

#include <cstdlib>
#include <iostream>

using namespace std;

template<class Policy_T>
struct ConstrainedValue
{
  public:
    // public typedefs
    typedef typename Policy_T policy_type;
    typedef typename Policy_T::value_type value_type;
    typedef ConstrainedValue self;

    // default constructor
    ConstrainedValue() : m(Policy_T::default_value) { }
    ConstrainedValue(const self& x) : m(x.m) {  }
```

www.TopSage.com

```
    ConstrainedValue(const value_type& x) { Policy_T::assign(m, x); }
    operator value_type() const { return m; }

    // uses the policy defined assign function
    void assign(const value_type& x) {
 Policy_T::assign(m, x);
    }

    // assignment operations
    self& operator=(const value_type& x) { assign(x); return *this; }
    self& operator+=(const value_type& x) { assign(m + x); return *this; }
    self& operator-=(const value_type& x) { assign(m - x); return *this; }
    self& operator*=(const value_type& x) { assign(m * x); return *this; }
    self& operator/=(const value_type& x) { assign(m / x); return *this; }
    self& operator%=(const value_type& x) { assign(m % x); return *this; }
    self& operator>>=(int x) { assign(m >> x); return *this; }
    self& operator<<=(int x) { assign(m << x); return *this; }

    // unary operations
    self operator-() { return self(-m); }
    self operator+() { return self(+m); }
    self operator!() { return self(!m); }
    self operator~() { return self(~m); }

    // binary operations
    friend self operator+(self x, const value_type& y) { return x += y; }
    friend self operator-(self x, const value_type& y) { return x -= y; }
    friend self operator*(self x, const value_type& y) { return x *= y; }
    friend self operator/(self x, const value_type& y) { return x /= y; }
    friend self operator%(self x, const value_type& y) { return x %= y; }
    friend self operator+(const value_type& y, self x) { return x += y; }
    friend self operator-(const value_type& y, self x) { return x -= y; }
    friend self operator*(const value_type& y, self x) { return x *= y; }
    friend self operator/(const value_type& y, self x) { return x /= y; }
    friend self operator%(const value_type& y, self x) { return x %= y; }
    friend self operator>>(self x, int y) { return x >>= y; }
    friend self operator<<(self x, int y) { return x <<= y; }

    // stream operators
    friend ostream& operator<<(ostream& o, self x) { o << x.m; return o; }
    friend istream& operator>>(istream& i, self x) {
 value_type tmp; i >> tmp; x.assign(tmp); return i;
    }

    // comparison operators
    friend bool operator<(const self& x, const self& y) { return x.m < y.m; }
    friend bool operator>(const self& x, const self& y) { return x.m > y.m; }
    friend bool operator<=(const self& x, const self& y) { return x.m <= y.m; }
    friend bool operator>=(const self& x, const self& y) { return x.m >= y.m; }
    friend bool operator==(const self& x, const self& y) { return x.m == y.m; }
    friend bool operator!=(const self& x, const self& y) { return x.m != y.m; }
  private:
    value_type m;
};

template<int Min_N, int Max_N>
struct RangedIntPolicy
{
  typedef int value_type;
  const static value_type default_value = Min_N;
  static void assign(value_type& lvalue, const value_type& rvalue) {
    if ((rvalue < Min_N) || (rvalue > Max_N)) {
 throw range_error("out of valid range");
    }
    lvalue = rvalue;
  }
};

#endif
```

www.TopSage.com

大家网 TopSage.com

示例5-11中的程序描述如何使用ConstrainedValue类型。

示例5-11 使用constained_value.hpp

```
#include "constrained_value.hpp"

typedef ConstrainedValue< RangedIntPolicy<1582, 4000> > GregYear;
typedef ConstrainedValue< RangedIntPolicy<1, 12> > GregMonth;
typedef ConstrainedValue< RangedIntPolicy<1, 31> > GregDayOfMonth;

using namespace std;

void gregOutputDate(GregDayOfMonth d, GregMonth m, GregYear y) {
  cout << m << "/" << d << "/" << y << endl;
}

int main() {
  try {
    gregOutputDate(14, 7, 2005);
  }
  catch(...) {
    cerr << "whoops, shouldn't be here" << endl;
  }
  try {
    gregOutputDate(1, 5, 1148);
    cerr << "whoops, shouldn't be here" << endl;
  }
  catch(...) {
    cerr << "are you sure you want to be using a Gregorian Calendar?" << endl;
  }
}
```

示例5-11中程序的输出如下：

```
7/14/2005
are you sure you want to be using a Gregorian Calendar?
```

讨论

当你使用日期和时间时，限制值类型是非常相关的，因为许多与日期时间相关的值都是整型并且它的值都限制在一个特定的范围内（例如，月份必须是在[0，11]区间内，月份中的天数必须是在[0，30]的区间内）。手工去验证每个函数参数是否在一个特定的范围内是非常花时间的并且容易出错。仅仅想象一下你是否想给一个100万行的程序处理日期范围的错误做一个全局的改变。

当ConstrainedValue模板类和RangedIntPolicy模板类一起使用时会使得定义几个不同的当所赋的值不在它的范围内时抛出异常的类型变得容易。示例5-12给出了几个不同的例子告诉你如何使用ConstrainedValue类来定义新的自验证整型类型。

示例5-12 ConstrainedValue的其他更多用法

```
typedef ConstrainedValue< RangedIntPolicy <0, 59> > Seconds;
typedef ConstrainedValue< RangedIntPolicy <0, 59> > Minutes;
typedef ConstrainedValue< RangedIntPolicy <0, 23> > Hours;
typedef ConstrainedValue< RangedIntPolicy <0, 30> > MonthDays;
typedef ConstrainedValue< RangedIntPolicy <0, 6> > WeekDays;
typedef ConstrainedValue< RangedIntPolicy <0, 365 > > YearDays;
typedef ConstrainedValue< RangedIntPolicy <0, 51> > Weeks;
```

www.TopSage.com

ConstrainedValue模板类是一个基于策略设计的例子。一个策略是一个作为模板参数传递的类，它指明了参数类型的行为或实现的某个方面。传递给ConstrainedValue的策略提供了如何在一个类型的同一个特例化之间的赋值实现细节。

使用策略能够把设计决定推迟到类的用户那里从而提高类的灵活性。当一组类有一个共同的接口但有不同的实现时使用策略是经常的。对一个给定的类，当不可能预见到某种可能的用法但又需要满足可能的使用时，策略特别有用。

还有很多别的策略类型可以与ConstrainedValue类一起使用。例如，无需抛出异常，你可以选择给它赋一个默认的值，或者赋一个最近的合法值。更进一步，限制可以不必总是一个范围：你甚至可以限制一个值总是偶数。

www.TopSage.com

第6章

数据管理

6.0 概述

本章将描述标准类库中一些你可以用来存储数据的数据结构。总的来说，它们都可称为容器，因为它们都能包含对象并且你能往这些容器中加入对象。本章也会描述另外一类不包含在标准类库中的容器，尽管它们和大部分的标准类库的实现打包在一起，这类容器称为哈希容器。

组成容器的这部分类库称为标准模板类库，或者STL，因为在它被包含进C++标准之前就是这样命名的。STL不仅包含本章要讲的容器，而且包含迭代器和算法，这两部分是STL的另外两个大的主题，它们使得STL成为了一个灵活的通用类库。因为本章主要讨论标准容器而不是STL本身，所以就像在很多的C++文献中一样我这里的容器也是指标准容器而不是STL容器。尽管我在这里也将讨论一些必要的迭代器和算法，但更多细节还要在第7章讨论。

C++标准使用精确的术语来描述它的容器集合类。C++标准中的容器是指标准中定义好接口的数据结构。例如，任何C++标准类库中的类都被称为容器并且一定要支持一个没有参数的`begin`成员函数，并且这个`begin`成员函数要返回一个指向容器中第一个元素的迭代器。C++标准中容器类定义了一系列必须的构造函数和成员函数。同时又定义了一些可选择的成员函数，仅仅是某些容器才实现它，常常这些函数能被高效地实现。

所有的容器集合可进一步分成两个不同类的容器：序列容器和联合容器。序列容器（通常叫做序列）按用户指定的顺序存储对象，并且提供了一个给定的接口（容器需求之外的）来访问和操作元素。联合容器有序地存储它的元素，因此它不允许你在特定的位置插入元素，尽管为了提高插入元素的效率，你可以提供一定的提示。序列容器和联合容器都需要支持一个必须的接口，但只有序列容器具有一个附加的操作集，并且这个操作

www.TopSage.com

集中的操作实现的效率更高。这些附加的序列操作比这个必须的接口提供了更好的灵活性和方便性。

这听起来很像继承。一个序列是一个容器，一个联合容器也是一个容器，但容器并不是序列，也不是联合容器。尽管从C++的角度来说，它不是继承，但从概念上来说，它是继承。一个矢量（vector）是一个序列，但它是一个单独的类；它不是继承自容器或其他类似的类（对于如何去实现矢量和其他的容器类，标准类库允许在实现上有不同的选择，且标准也不强制标准类库的实现一定要包含一个容器基类）。在容器的设计上是百花齐放，你如果有兴趣了解更多的内容，可以参考Matt Austern著的《泛型编程和STL》(Addison Wesley)。

这一章分为两部分。第一部分描述了如何使用矢量，矢量是一个标准的序列，也是非常流行的数据结构之一。这一部分描述了如何有效和高效地使用矢量。其余部分讨论了其他一些使用得比较广的标准容器类，包括前面提到过的两个非标准的哈希容器。

6.1 使用矢量而不是数组

问题

如何在一个序列中存储一些值（内置类型，对象，指针等），同时又可以随机访问其中的元素，并且又不能静态地决定序列的大小？

解决方案

使用定义在<vector>中的矢量模板类，不要使用数组。矢量看上去像是数组，但它比数组具有更多安全和方便的优点。示例6-1给出了矢量的一些通常的操作。

示例6-1 使用矢量的普通成员函数

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

int main() {

   vector<int>    intVec;
   vector<string> strVec;

   // Add elements to the "back" of the vector with push_back
   intVec.push_back(3);
   intVec.push_back(9);
   intVec.push_back(6);

   string s = "Army";

   strVec.push_back(s);
   s = "Navy";
   strVec.push_back(s);
```

www.TopSage.com

```
    s = "Air Force";
    strVec.push_back(s);

    // You can access them with operator[], just like an array
    for (vector<string>::size_type i = 0; i < intVec.size(); ++i) {
 cout << "intVec[" << i << "] = " << intVec[i] << '\n';
    }

    // Or you can use iterators
    for (vector<string>::iterator p = strVec.begin();
    p != strVec.end(); ++p) {
 cout << *p << '\n';
    }

    // If you need to be safe, use at() instead of operator[].  It
    // will throw out_of_range if the index you use is > size().
    try {
 intVec.at(300) = 2;
    }
    catch(out_of_range& e) {
 cerr << "out_of_range: " << e.what(  ) << endl;
    }
}
```

讨论

总的来说，如果你需要使用数组的话，你都可以用矢量来代替。矢量比数组提供了更好的安全性和灵活性，并且在大多数情况下，性能上的影响都是可以忽略不计的，一般你都能发现它的性能比你想象的更好，通过一些成员函数你还能调优矢量的性能。

如果你不熟悉标准类库中的容器，或者不熟悉使用类模板，示例6-1中说明的矢量就会需要一些解释。一个矢量的声明看上去和下面一样：

```
vector<typename Value, // The type of element this vector will hold
  typename Allocator = allocator<Value> > // The memory allocator
                                          // to use
```

标准容器需要你传给它一个你希望它存储对象的类型作为参数。内存分配函数allocator也需要这个模板参数，但它通常都有一个默认的值，通常不应改变这个默认的参数值，这里就不讨论它。

如果你希望用一个矢量来存储整型值，你可以与下面一样来声明：

```
vector<int>    intVec;
```

如果你希望用一个矢量来存储字符串，你只用改变这个矢量的类型参数：

```
vector<string> strVec;
```

矢量能存储任何的C++类型，只要这个类型支持复制构造函数和赋值操作。

当你初始化了一个矢量后，接下来的操作就是把你希望存储的对象追加到它里面去。可以使用push_back函数来把元素加到它的尾部：

```
intVec.push_back(3);
intVec.push_back(9);
intVec.push_back(6);
```

www.TopSage.com

大家网 TopSage.com

上面基本上等价于向一个数组追加了元素0、1和2。说它是基本等价是因为push_back是个成员函数，它把它的参数加到矢量的尾部并返回一个无类型的值。操作符[]返回数组中索引所对应的内存位置。push_back函数能保证矢量中的内置缓存具有足够的大小，如果这个缓存中还有可用的空间，那它就把元素加到下一个还没使用的索引对应的位置上，如果这个内置的缓存没有足够的空间，它就按照已经定义好的算法来扩充这个缓存，然后再把元素加入进去。

你也可以使用insert成员函数在一个矢量的中间插入元素，尽管你应该避免这样做，因为这样的插入所花的时间是一个线性的复杂度。请参考6.2节中关于使用矢量的性能问题的详细分析。为了插入一个元素，你需要得到一个指向你想插入位置开始处的迭代器（请参考7.1节中迭代器的讨论）：

```
string s = "Marines";
vector<string>::iterator p = find(strVec.begin(  ),
    strVec.end(  ), s);

if (s != strVec.end(  ))    // Insert s immediately before the element
   strVec.insert(p, s);     // p points to
```

insert重载函数能够让你在一个矢量中插入一个对象的多个副本，以及来自另一个序列的所有元素（这个序列可以是另一个矢量、数组、链表等）。

无需使用插入操作，你可能想简单地给一个矢量赋一个来自于别的地方的现有序列，从而把这个矢量中原有的值清除。assign成员函数可以完成这样的功能。你能够和下面一样给一个矢量赋一系列的值，或同一个对象的多个副本：

```
string sarr[3] = {"Ernie", "Bert", "Elmo"};
string s = "Oscar";
strVec.assign(&sarr[0], &sarr[3]); // Assign this sequence
strVec.assign(50, s);              // Assign 50 copies of s
```

如果新的序列比原来的序列缓存大的话，assign函数会自动调整缓存的大小从而把这个新的序列存储起来。

一旦你把数据存到一个矢量中后，你就有很多办法把数据取出来。大概最直观的办法是操作符[]，依据你调用的矢量是否是常量型，返回一个在这个索引处的元素的引用或常量引用。在这个方面，更像一个数组：

```
for (int i = 0; i < intVec.size(  ); ++i) {
   std::cout << "intVec[" << i << "] = "
   << intVec[i] << '\n'; // rvalue
}
intVec[2] = 32; // lvalue
```

你如果使用了一个比矢量中最后一个元素的索引还大的索引来访问矢量的话，操作符[]的行为就会像在数组中一样，给你返回一个没定义的值，这常常使得你的程序崩溃或破坏你的数据。通过使用size()查询矢量中元素的个数，你能避免这种情况的出现。你应该优先使用迭代器而不是操作符[]，因为在任何标准的容器中，遍历元素，使用迭代器是一个传统的方法：

www.TopSage.com

```
for (vector<string>::iterator p = strVec.begin();
p != strVec.end(); ++p) {
   std::cout << *p << '\n';
}
```

迭代器是最有用的方法，因为它能让你用更为一般的方法与容器交互。例如，如果你写了在两个迭代器之间的一序列元素上操作的算法，那么它就能应用到任何标准容器上。这就是泛型方法。你如果用操作符[]来随机访问元素，你就限制这个算法只能应用到支持这个随机访问的这类容器上。前一个通过迭代器的方法就是定义在<algorithm>中的标准类库算法能与标准容器（或其他与这些容器具有类似工作方式的容器类）无缝对接地工作的方法。

矢量也提供了标准数组不能提供的安全性。不像数组，矢量在它的成员函数中提供了越界检查。你如果用一个非法的索引来访问矢量的话，它将抛出一个out_of_range异常，这样你就能捕获这个异常并采取一定的措施。例如：

```
try {
   intVec.at(300) = 2;
}
catch(std::out_of_range& e) {
   std::cerr << "out_of_range: " << e.what() << std::endl;
}
```

正如你所知道的那样，你如果用一个数组末端后面的索引来使用操作符[]来引用数组中的元素，这个操作符将会根据你告诉它的那样工作，把这个索引所对应的元素的内存位置返回给你。这是不太好的，因为这样要么因为访问了不该访问的内存而导致程序崩溃，要么程序更新那些属于别的堆对象的内存，这样常常更坏。矢量中操作符[]也是这样的工作方式，在你需要安全性的时候，最好最低限度地使用它。

这是矢量导致程序崩溃的方面。但什么是矢量？你如果正在用C++写程序，你可能意识到性能问题，并且不想就这么简单地学到这些。不用着急。6.2节将要讨论矢量的工作方式并会给出一些有效使用它的一些技巧。请参考6.2节。

6.2 高效地使用矢量

问题

如何高效地使用矢量？你有空间或时间上的限制，所以需要减少或消除性能上的问题。

解决方案

理解矢量如何实现的，知道插入和删除成员函数的复杂性，并用reserve成员函数来减少不必要的内存抖动。示例6-2通过一个简单的例子来说明这些技术。

示例6-2 高效地使用矢量

```
#include <iostream>
#include <vector>
#include <string>
```

www.TopSage.com

```
using std::vector;
using std::string;

void f(vector<string>& vec) { // Pass vec by reference (or
                              // pointer, if you have to)
   // ...
}

int main() {

   vector<string> vec(500); // Tell the vector that you plan on
                            // putting a certain number of objects
                            // in it at construction
   vector<string> vec2;

   // Fill up vec...
   f(vec);

   vec2.reserve(500);  // Or, after the fact, tell the vector
                       // that you want the buffer to be big
                       // enough to hold this many objects

   // Fill up vec2...
}
```

讨论

高效使用矢量的关键是要知道它们是如何工作的。一旦你对一些矢量通常的实现有很好的了解，性能问题就变得很好解决。

矢量的工作原理

从本质上来说，一个矢量就是一个管理好的数组。更特殊的说，一个vector<T>矢量就是一块连续的内存（例如，数组），这块连续的内存足够大，能够装下n个类型T的对象，这里n是一个大于等于0的整数并且小于或等于系统定义好的最大容量值。n通常在容器的整个生命周期内可以增大，因为你可以给容器增加和减少元素，但通常n不会减小。使得矢量和数组不同的是矢量在它的内存块上、插入和获取元素的成员函数上的自动内存管理，并且矢量的成员函数还能提供矢量的一些元数据，如矢量的大小（有多少个元素）和矢量的最大容量（缓存的大小），同时还能提供一些类型信息：vector<T>::value_type是指T类型、vector<T>::pointer是一个指向T类型的指针等。最后的这两个以及其他的几个都是标准容器提供的，这使得你能写出泛型代码从而不用考虑T的类型。图6-1给出矢量的一些成员函数，假设这个矢量有7个元素且容量是10。

你如果对标准类库厂商是如何实现矢量感兴趣的话，请编译一下示例6-1并且对每个矢量成员函数调用单步跟踪进去，或者打开你的标准类库实现的<vector>头文件看一看。也许这些代码不是你看到过的读起来最友好的代码，但是它能给你一些思路。首先，如果你以前从没看过库的代码的话，它能让你学到该如何去写高效的、可移植的泛型代码的技术；第二，它能让你了解到你正在使用的任何容器的实现。你如果正在写一些需要在不同的标准类库实现上运行的代码，你也应使用它的这种方式来写代码。

www.TopSage.com

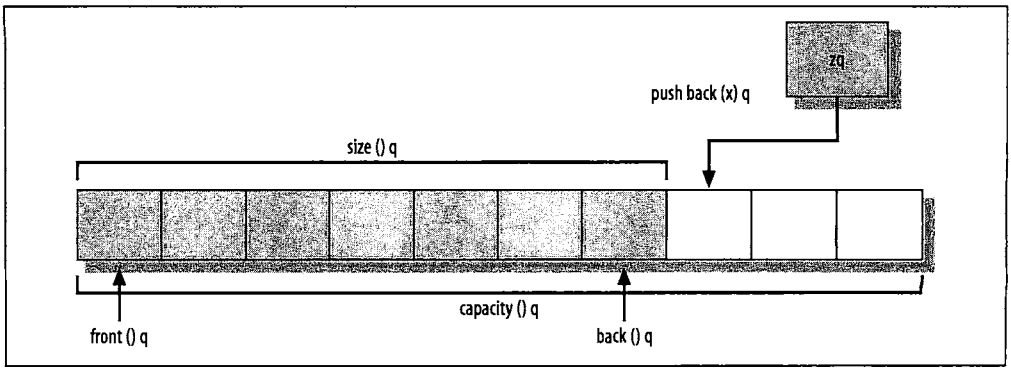


图6-1 一个矢量的内部结构

尽管各个类库实现的厂商不同，但大部分矢量的实现基本相同。都有一个成员变量，它指向一个T类型的数组，你追加或分配的元素都是通过复制构造函数或赋值到每个矢量的元素上的。

把一个T对象加到缓存中是通过放置new操作实现的，这样说是因为你给了复制构造函数这个将要被构造出来的对象的类型和这个将要被构造出来的对象应在哪里被构造出来的内存地址信息——在缓存中的下一个可能的内存块中被构造。你如果使用索引（操作符[]或操作符at）来明显地指出内存块来赋值的话，那么T类型的赋值操作符就被直接使用了。请注意，在这两种情况下，你的对象被复制构造函数或T的赋值操作符T::operator=克隆了。这个矢量不是简单地储存你加入的对象的地址。就是因为这个原因，任何可以存储在矢量当中的类型都是可以复制构造和赋值的。这个属性也就意味着通过T类型的复制构造函数和赋值操作符，一个新的T类型对象就被构造了。由于矢量的复制入和复制出的语义，这点是很重要的——如果复制构造或赋值一个你的对象不能正确地运行，那么你从矢量取到的元素和你加到矢量中的元素就可能不同。这就会产生非常严重的错误。

一旦你在一个矢量中加入了一序列的元素后，它的缓存就会被占满了，这时缓存就必须被扩充以容纳任何新的元素。缓存增长的算法是预先实现好的，但常常是按$2n+1$容量增长的。这里重要的概念是矢量如何增长它的缓存。你不可能用一个不确定大小的块来告诉操作系统扩充你的堆内存；当你需要的内存块比你使用过的内存块大时，你就不得不申请一块新的内存。这个缓存增长的过程如下：

1. 为一个新的缓存分配内存。
2. 把旧的缓存中的数据复制到这个新的缓存中。
3. 删除这个旧的缓存。

这就使得矢量能把它所有的对象保存在一块连续的内存中。

www.TopSage.com

优化矢量的性能

前几部分主要讨论了对象在矢量中是如何存储的。从这个角度讲，也许你已经意识到了矢量的主要性能问题所在了，也许你还没有意识到，下面就是这个问题的讨论。

作为开始，一个矢量（或别的标注类库容器）不存储你的对象，而是你对象的副本。也就是说每次你把对象加到矢量中去，你并不是真正把你的对象本身加到矢量中；而是通过了复制构造函数或赋值操作符把你的对象复制一个放到矢量中。类似地，当你从一个矢量中取出对象时，你也是把矢量中某个元素复制到你的本地变量中。考虑如下一个简单地从矢量中复制元素的例子：

```
vector<MyObj> myVec;
// Put some MyObj objects in myVec
MyObj obj = myVec[10];  // Copy the object at index 10
```

因为myVec[10]是右值，所以这个赋值用myVec[10]返回的对象作为参数调用obj的赋值操作符。当对象非常多时，来自对象复制的开销将增加得非常快，所以最好要避免这样用。

在矢量中存储指针而不是对象本身能够减少复制的成本。存储指针的话，向矢量中加入数据和从矢量中取数据都需要更少的CPU周期，因为复制指针比复制对象本身更快，同时矢量缓存需要的内存也更少。但必须记住的是，如果你把指针加到一个标准类库的容器中，当容器被破坏时，容器并不删除它的指针所指向的对象。容器只是破坏掉它所包含的对象，例如，一个变量它存储一个指向一个对象的指针，但容器并不知道它存储的是指针还是对象，它所知道的仅是类型T的对象。

调整缓存的大小代价也不小。复制缓存中的每个元素要花很多的工作，所以这种操作就应尽量避免。为了避免这种情况的发生，需要显式指明缓存的大小。有许多方法可达到这个目的。最简单的办法就是在构造一个矢量时指明：

```
vector<string> vec(1000);
```

这将保存1000个字符串的空间并且使用字符串的默认的构造函数来初始化这些字符串。使用这种方法，你要付出构造每一个这种元素的代价，但是通过给每个元素初始化为一个空的字符串增加了这个矢量的安全性。也就是你可以引用那些没有赋值的元素，你将简单地得到一个空的字符串对象。

如果你想把这个缓存初始化成某个特殊的值，你可以传递一个对象，这样你就能把缓存中的每个元素都复制进这个值。

```
string defString = "uninitialized";
vector<string> vec(100, defString);
string s = vec[50]; // s = "uninitialized"
```

通过这种形式，vec将从defString中复制进100个元素。

www.TopSage.com

大家网 TopSage.com

另一种方法是在矢量被构造出来后调用reserve成员函数：

```
vector<string> vec;
vec.reserve(1000);
```

调用reserve函数与在构造函数中指定大小这两种方法中，最大的不同是reserve函数不用任何值来初始化缓存中的元素。特别地，这也就意味着你不应该引用你还没有加入值的索引：

```
vector<string> vec(100);
string s = vec[50];       // No problem: s is now an empty string
vector<string> vec2;
vec2.reserve(100);
s = vec2[50];             // Undefined
```

使用reserve函数或在构造函数中指定默认的对象能使你避免不必要的缓存再分配。这将有助于性能的提高，同时又要注意另外一个问题：任何时候缓存如果被重新分配后，任何指向它原来的元素的迭代器都不可用了。

最后，除了在矢量的尾端插入元素之外的任何别的位置插入元素都是不好的。请回头看图6-1。由于一个矢量就是一个带有一些附加功能的数组，所以为什么只应在尾端插入元素就很好理解了。矢量中的元素是连续存储的，当你在别的任何位置而不是尾端，假设是索引n处，那么从$n+1$处到尾部的所有元素都必须向后移动一个位置才能给这个新元素腾出一个空间。这个操作的代价是线性的，也就是对于一个一般规模大小的矢量，这个操作的代价也是很高的。在矢量中删除一个元素也是如此：索引比它大的那些元素都要向前移动一个位置。如果你需要在容器的任何别的位置而不是末端插入和删除元素的话，你应该使用链表而不是矢量。

6.3 矢量复制

问题

如何把一个矢量中的内容复制到另一个矢量中去？

解决方案

有很多办法可以用来完成这个目的。当你创建一个矢量时你可以使用复制构造函数，或者使用assign成员函数。示例6-3说明了是如何这样做的。

示例6-3 复制矢量中的内容

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

// Util function for printing vector contents
template<typename T>
```

www.TopSage.com

```
void vecPrint (const vector<T>& vec) {
   cout << "{";
   for (typename vector<T>::const_iterator p = vec.begin();
  p != vec.end(); ++p) {
 cout << "{" << *p << "} ";
   }
   cout << "}" << endl;
}

int main() {

   vector<string> vec(5);
   string foo[] = {"My", "way", "or", "the", "highway"};

   vec[0] = "Today";
   vec[1] = "is";
   vec[2] = "a";
   vec[3] = "new";
   vec[4] = "day";

   vector<string> vec2(vec);
   vecPrint(vec2);

   vec.at(0) = "Tomorrow";

   vec2.assign(vec.begin(), vec.end()); // Copy each element over
   vecPrint(vec2);                      // with assign

   vec2.assign(&foo[0], &foo[5]); // Assign works for anything that
   vecPrint(vec2);                // behaves like an iterator

   vector<string>::iterator p;

   p = find(vec.begin(), vec.end(), "new");

   vec2.assign(vec.begin(), p); // Copy a subset of the full range
   vecPrint(vec2);              // of vec
}
```

讨论

复制一个矢量很容易，有两种办法可用。你可以使用复制构造函数把一个矢量中的内容复制到另一个中，就像任何其他对象那样，或者使用assign成员函数。有一点想说的就是关于复制构造函数，仅仅只用把你要克隆的矢量作为参数传给它就可以了。如下所示：

```
vector<string> vec2(vec);
```

在这个例子中，矢量vec2将拥有与矢量vec同样多的元素，并且这些元素中的每一个都是vec矢量中对应元素的副本。每一个元素都是通过字符串的复制构造函数构造出来的。由于这是一个构造过程，所以矢量vec2的缓存一定不会比vec的缓存小。

assign成员函数工作方式基本相同，但还是有点差别，因为这个成员函数工作对应的目标矢量可能包含数据。所以，首先，这个矢量中的数据对象必须先被销毁掉，assign成员函数是调用这些数据对象的析构函数达到这个目的的。一旦这个目标矢量中原有的对象没有了，它就要检查vec2矢量的缓存大小以保证它能够装下vec矢量中的所有对象。如果发现vec2的缓存太小，assign成员函数就必须调准缓存的大小以使得它能把这些新的对象都能容纳得下。最后，assign成员函数再把每个对象复制过来。

www.TopSage.com

此外，你还可以使用assign成员函数复制一个序列中的部分元素。例如，你如果只想用assign成员函数复制矢量vec中的一部分元素，你就可以给它指定你想复制的范围给assign成员函数：

```
vector<string>::iterator p;
p = std::find(vec.begin(), vec.end(), "new");
vec2.assign(vec.begin(), p);
vecPrint(vec2);
```

这个例子中，assign成员函数将复制从开始到p指向的所有元素，但不包含p指向的元素。这是因为，就像在所有的标准类库容器和算法中那样，assgin(first, last)复制从first指向的元素到last指向的所有元素，但不包含last指向的元素。像这样只包含first指向的元素和从first到last指向的所有元素但不包含last指向的元素常常可表示为*[first, last)*。

使用assign或复制构造函数而不是遍历整个矢量本身。也就是说，不要去遍历矢量vec来复制每个元素然后再把复制出来的对象加到vec2矢量中去。因为这样你要写的代码更多（冗余），并且这样不可能利用到你的标准类库实现中的assign成员函数或复制构造函数的任何优化功能。

6.4 将指针存储在矢量中

问题

为了效率或其他原因的考虑，你不能在一个矢量中存储你的对象的副本，但你又需要能在一定程度上跟踪到它们，如何实现？

解决方案

在矢量中存储对象的指针而不是对象本身。但如果这样做的话，不要忘记删除这些对象，因为矢量是不能为你删除它们的。示例6-4声明了一个存储指针的矢量并且给出了如何使用它的方法。

示例6-4 使用一个存储指针的矢量

```
#include <iostream>
#include <vector>

using namespace std;

static const int NUM_OBJECTS = 10;

class MyClass { /*...*/ };

int main() {

   vector<MyClass*> vec;

   MyClass* p = NULL;
```

www.TopSage.com

```
    // Load up the vector with MyClass objects
    for (int i = 0; i < NUM_OBJECTS; i++) {
       p = new MyClass();
       vec.push_back(p);
    }

    // Do something useful with this data, then delete the objects when
    // you're done
    for (vector<MyClass*>::iterator pObj = vec.begin();
         pObj != vec.end(); ++pObj) {
       delete *pObj; // Note that this is deleting what pObj points to,
                     // which is a pointer
    }

    vec.clear(); // Purge the contents so no one tries to delete them
                 // again
}
```

讨论

你可以在矢量中存储指针，就像用它存储任何别的对象一样。声明一个存储指针的矢量如下：

```
vector<MyClass*> vec;
```

需要记住的重要一点就是当矢量存储对象时，矢量本身不知道它存储的对象所代表的是什么值。因此，当矢量自身被清除时，它不会去删除它存储的指针所指的对象。如果你分配了一块内存给某个指针，然后把这个指针加到一个矢量中，当你清除这个矢量时，你将不得不亲自把这块内存释放掉。不要被“容器”这个字眼所蒙蔽就以为你在矢量中存储指针时，这个矢量就拥有了这个指针的一切。

当你删除了指针所指向的元素后，就像你要把指针变量设为NULL一样，你也应该显式地把矢量清空。这将能防止这个指针再被错误地删除。

6.5 将对象存储在链表中

问题

如何把元素存储到一个序列中？矢量并不能很好地满足你的要求。特别地，你需要能够有效地在序列的中间追加和删除元素，而不是仅在序列的尾部。

解决方案

使用定义在<list>中的链表来存储数据。当在序列中除了开始处和结束处的其他地方改变序列时，链表提供了更好的性能和更好的灵活性。示例6-5说明了如何使用链表，并且说明了它的一些独特操作。

示例6-5 使用链表

```
#include <iostream>
#include <list>
#include <string>
#include <algorithm>
```

www.TopSage.com

```
using namespace std;

// A simple functor for printing
template<typename T>
struct printer {
   void operator()(const T& s) {
 cout << s << '\n';
   }
};

bool inline even(int n) {
   return(n % 2 == 0);
}

printer<string> strPrinter;
printer<int>    intPrinter;

int main() {

   list<string> lstOne;
   list<string> lstTwo;

   lstOne.push_back("Red");
   lstOne.push_back("Green");
   lstOne.push_back("Blue");

   lstTwo.push_front("Orange");
   lstTwo.push_front("Yellow");
   lstTwo.push_front("Fuschia");

   for_each(lstOne.begin(), // Print each element in the list
  lstOne.end(),             // with a custom functor, print
  strPrinter);

   lstOne.sort();           // list has a member for sorting
   lstTwo.sort();

   lstOne.merge(lstTwo);     // Merge the two lists and print
   for_each(lstOne.begin(),  // the results (the lists must be
  lstOne.end(),              // sorted before merging)
  strPrinter);

   list<int> intLst;

   intLst.push_back(0);
   intLst.push_back(1);
   intLst.push_back(2);
   intLst.push_back(3);
   intLst.push_back(4);

   // Remove all values greater than 2
   intLst.remove_if(bind2nd(greater<int>(), 2));

   for_each(intLst.begin(),
  intLst.end(),
  intPrinter);

   // Or, remove all even values
   intLst.remove_if(even);
}
```

讨论

一个链表是一个序列，它给在任何位置上插入或者删除元素提供了一个常量的复杂度，但是查找元素的复杂度则是线性的。链表常常实现为双向链表，这就意味着存在每一个

www.TopSage.com

大家网 TopSage.com

元素对应的节点都有一个指向序列中它前面节点和一个指向它后面节点的指针。它满足了一个标准序列容器的所有需求，并且提供了一些独特的成员函数。

声明一个链表是很简单的，仅需给定你需要存在这个链表中元素的类型，并且可选地指定一个内存分配的类：

```
list<typename Value, // The type of element stored in the list
typename Allocator = allocator<Value> > // The memory allocator
                                        // to use
```

这个Value模板参数就是需要存储在这个链表中元素的类型。这个类型必须是支持复制构造和赋值操作符的类型。Allocator是将要使用的内存分配类；并且标准的分配类是默认的（大部分情况下，这个默认的分配类都能满足）。

下面就是一个典型的链表声明（来自示例6-5）：

```
list<string> lstOne;
```

当你声明了你的链表后，你就可以使用push_front或者push_back来把元素加到它里面去，如下所示：

```
lstOne.push_back("Red");   // Add these to the end of the list
lstOne.push_back("Green");
lstOne.push_back("Blue");

lstTwo.push_front("Orange");  // Add these to the beginning
lstTwo.push_front("Yellow");
lstTwo.push_front("Fuschia");
```

把元素加入一个链表需要花费常数时间，但不是与一个矢量那样是平摊常量时间。一个链表的实现不需要偶尔调整它的缓存区的大小，因此你就不会如矢量那样不时地感到性能上的降低。链表仅是需要更新少数指针而已。

使用pop_front或者pop_back（无参数）可以从这个链表的开始处或末尾处删除元素。尽管这个名字是这样，但这个“pop”成员函数并不如你期望的（如典型的栈语义）那样返回弹出的元素。为了得到一个序列中开始处或末尾处的元素的引用，你需要使用back或者front成员函数。

典型地，一个链表就像在图6-2中显示的那样。每个节点都有（最少）三部分：它所包含的对象、一个指向它前面节点的指针和一个指向它后一个节点的指针。本节的后面我将使用next_和prev_来指代这两个指针。

一旦你明白了一个链表是如何实现的时，为什么它的不同操作比起矢量来具有不同的复杂度，其答案就显而易见了。在链表中的任何位置增加一个元素仅需要在它的前面和后面的节点对它们的next_和prev_指针作相应的调整。当你插入或删除一个元素时，链表的一个好的地方是只有指向这个受影响的对象的迭代器才会变成不合法。而指向别的元素的迭代器则不受影响。

www.TopSage.com

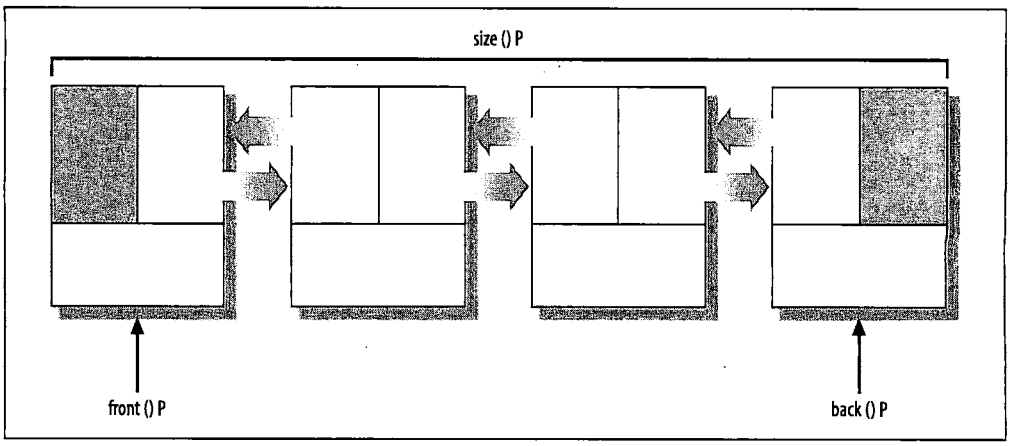


图 6-2 一个双向链表

插入和删除方法就是 insert 和 erase。insert 带一个迭代器作为它的第一个参数，而另一个就是 T 类型的对象，一个数字和一个 T 类型的对象，或者一个末端迭代器作为它的第二个参数。这个指向元素的迭代器指向这个需要插入的元素之前的那个元素。这个 insert 函数的重载可以被如下这样使用：

```
list<string> strLst;
list<string>::iterator p;
// ...
string s = "Scion";

p = find(strLst.begin(), strLst.end(),  // std::find from <algorithm>
    "Toyota");

strLst.insert(p, s);       // Insert s right before p
strLst.insert(p, 16, s);   // Insert 16 copies of s right before p
strLst.insert(p, myOtherStrLst.begin(),  // Insert everything in
    myOtherStrLst.end());                // myOtherStrLst before p
Erasing elements is similar:
p = find(strLst.begin(), strLst.end(),   // std::find from <algorithm>
    "Toyota");

strLst1.erase(p);   // Erase this element
strLst2.erase(p, strLst.end()); // Erase p to the end
strLst3.clear(); // Erase all elements
```

除了标准容器成员函数之外，链表也提供一些有趣的函数。第一个就是 splice 函数。

splice 的功能如下：它把两个链表接到一起。示例 6-5 中说明了把 lstTwo 链表接到 lstOne 中去：

```
list<string>::iterator p =  // Find somewhere to insert the other
  std::find(lstOne.begin(), // list
  lstOne.end(), "Green");
lstOne.splice(p, lstTwo);   // Insert lstTwo right before "Green"
```

p 是一个指向 lstOne 中元素的迭代器。lstTwo 被插入到 lstOne 中 p 的前面。就像是一个插入操作一样，这里所有需要做的就是改变这个受影响的节点的next_和prev_

www.TopSage.com

指针，因此这个操作也需要花费常量意义上的时间。当你把lstTwo接到lstOne上后，lstTwo就变成空了，这就是为什么lstTwo不是一个常量型的参数。你也能把来自lstTwo中的一个简单的元素插入到lstOne中去，或者把来自lstTwo中某个范围内的元素插入到lstOne中去。在这两种情况下，这些接到lstOne中的元素都会被从它的原始链表中删除。

如果你的链表是排过序的(链表有它自己的排序成员函数；std::sort不能与一个链表协同工作)，并且你需要合并链表并保持它们的顺序，你就可以使用merge而不是splice方法。merge将会把两个链表合并成一个，并且如果两个元素相等的话，来自lstOne的元素将会在这个最后的链表中排在前面。就像在splice方法中一样，这个参数链表然后就会变成空。

链表为删除元素也具有一些很好的集合操作。假设你想删除某个元素的所有可能的出现。你需要做的就是用一个参数调用remove方法，这个参数将会与这个链表中每一个项目做比较，也即是给出这个(*p == item) != false的值，这里p是一个链表的迭代器。可以如下这样调用remove方法：

```
strLst.remove("Harry");
```

这行代码将会从这个strLst中删除那些等于“Harry”的元素。如果你需要删除那些满足其他断言的元素，如那些比某个值大的元素，就可以使用remove_if方法：

```
bool inline even(int n) {
   return(n % 2 == 0);
}

list<int> intLst;
// Fill up intLst...
intLst.remove_if(even);  // Removes all elements where even(*p)
                         // is != false
```

如果你的断言非常复杂，就考虑使用那些来自<functional>中的算符。例如，如果你想删除一些比某个值大的元素，你就可以与remove_if一起使用greater（来自<algorithm>）和bind2nd：

```
intLst.remove_if(std::bind2nd(std::greater<int>(), 2));
```

这将删除intLst中比2大的所有元素。这个语法还是有点深奥的，但是它的结果还是易懂的。bind2nd带两个参数，一个函数对象（称之为f）和一个值（v），并且返回一个函数对象，这个函数对象带一个简单的参数（arg）并且调用f(arg, v)。对于不使用很多的函数而完成这类工作来说bind2nd是一个聪明的方法。

当你需要一个标准的序列容器时，链表是矢量的好的代替品。链表的不同内部表示给许多标准的序列操作提供了不同的复杂度和一些它自己的有趣操作。

请参考6.1节。

www.TopSage.com

6.6 将字符串映射到其他事物

问题

你有一些对象需要存储到内存中，并且你需要使用它们的字符关键字来存储它们。如何能够快速地追加、删除和获取它们（最多是对数级别的复杂度）？

解决方案

使用标准容器map，它声明在<map>中，它把关键字（字符串）映射成值（任何遵循它的语义的类型）。示例6-6说明是如何这样做的。

示例6-6 创建一个字符串map

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

int main() {

   map<string, string> strMap;

   strMap["Monday"]    = "Montag";
   strMap["Tuesday"]   = "Dienstag";
   strMap["Wednesday"] = "Mittwoch";
   strMap["Thursday"]  = "Donnerstag";
   strMap["Friday"]    = "Freitag";
   strMap["Saturday"]  = "Samstag";
   // strMap.insert(make_pair("Sunday", "Sonntag"));
   strMap.insert(pair<string, string>("Sunday", "Sonntag"));

   for (map<string, string>::iterator p = strMap.begin();
 p != strMap.end(); ++p ) {
    cout << "English: " << p->first
    << ", German: " << p->second << endl;
   }

   cout << endl;

   strMap.erase(strMap.find("Tuesday"));

   for (map<string, string>::iterator p = strMap.begin();
 p != strMap.end(); ++p ) {
    cout << "English: " << p->first
    << ", German: " << p->second << endl;
   }
}
```

讨论

一个map是一个用来把关键字映射成值的联合容器，它给插入和查找操作提供了对数级的复杂度，而给删除简单元素提供了常数时间。对开发者来说，使用一个map来使用一个字符串关键字来跟踪对象是非常普遍的。这就是示例6-6所做的；在这个例子中，这个映射的类型碰巧也是一个字符串，但是它可以是任何类型。

www.TopSage.com

一个 map 可以如下这样声明：

```
map<typename Key,                              // The type of the key
    typename Value,                            // The type of the value
    typename LessThanFun = std::less<Key>,     // The function/functor
                                               // used for sorting
    typename Alloc = std::allocator<Key> >     // Memory allocator
```

Key和Value是将要存储在这个map中的键和其关联值对应的类型。LessThanFun是一个带两个参数的函数或算符，并且如果第一个比第二小的话就返回真值；默认的情况下使用标准的less算符。Alloc则是内存分配器，默认的情况下是这个标准的allocator分配函数。

使用 map 是非常简单的。你可以如下这样来声明键和值的类型：

```
map<string, string> strMap;
```

这就创建了一个map，它的健和值都是字符串类型。你可以使用操作符[]来向这个map中加入对象，这种方法是非常直接的并且很容易读懂：

```
strMap["Monday"]    = "Montag";
strMap["Tuesday"]   = "Dienstag";
strMap["Wednesday"] = "Mittwoch"; // ...
```

这将使用一个下标（例如，“Monday”）作为关键字和等号右边的值作为值来给这个map插入元素。如果你提供了LessThanFun模板参数的话，它们将基于这个LessThanFun模板参数来给这些元素排序；如果你没有提供的话，就是用默认的std::less<Key>来排序。

为了从一个 map 中得到值，你可以在一个赋值的右边使用操作符[]，如下所示：

```
wedInGerman = strMap["Wednesday"];
```

按照所有的标准容器的方式，这个与关键字 "Wednesday" 相对应的值被使用操作符 = 复制进对象 wedInGerman 中。

对于插入或更新或者是获取一个map中的元素，操作符[]是一个方便的方法，但是它有一个微妙的行为可能不是你所期望的。严格地说，操作符operator[k]返回一个与这个关键字k对应的值的索引，无论这个关键字k是否存在于这个map中。如果关键字k存在于这个map中，那么就返回与之对应的值。如果不存在的话，这个关键字k就被插入到这个map中并且值类型的默认构造函数就会被使用来给这个关键字k创建一个值对象。为了使得这个过程更具体，考虑如下的这段代码：

```
map<string, string> mapZipCodes;  // There are zero elements now

string myZip = mapZipCodes["Tempe"];  // Nothing is in the map yet,
                                      // but what is count() now?
```

myZip中有什么呢，并且现在在mapZipCodes中有多少个元素呢？由于关键字如果不存在这个 map 中的话，操作符[]将会把它插到这个 map 中，那么 myZip 就会是一个空

www.TopSage.com

的字符串并且在mapZipCodes中就会有一个元素。这个行为可能就不是你所期望的，但是无论它是否是你期望的，你都应意识到操作符[]不是一个常量型的成员函数：永远都有可能由于增加一个节点而改变这个map的状态。

这个insert成员函数给向一个map增加键/值对提供了一个替代方案。insert函数执行一个严格的插入操作，而不是和操作符[]那样执行一个插入更新操作。如果你使用一个map（而不是multimap，它提供了多个关键字），如果这个关键字在这个map中已存在，则insert成员函数将不会做任何事情。相反，操作符在关键字在map中存在的情况下会用你的值对象代替原来的值对象。

但是insert函数的语法需要的工作比操作符[]的更多，并且它与map是如何存储数据相关的。考虑来自示例6-6中的如下这行代码：

```
strMap.insert(std::make_pair("Sunday", "Sonntag"));
```

这个map在一个pair对象中存储键/值对。一个pair是一个简单的类模板（定义在<utility>中并且被<map>包含），它拥有两个类型的两个值。为了声明一个字符串pair，你可以如下这样做：

```
pair<string, string> myPair;
```

在这个pair对象中的第一个和第二个元素都可以通过公共的成员函数first和second访问。如果你使用操作符[]来访问一个map中的元素的话，那么你就不可以常常不得不直接处理pair对象，但是使用许多其他的成员函数你就可以，因此知道如何创建和引用一个pair对象是很好的。例如，迭代器则是简单地间接引用了一个pair对象，因此当你使用它们时，就像我在示例6-6中那样，你应该首先知道如何获取关键字和它对应的值。

```
for (map<string, string>::iterator p = strMap.begin();
   p != strMap.end(); ++p )
      cout << "English: " << p->first
           << ", German: " << p->second << endl;
```

关键字存储在first元素中，而对应的值则存储在second元素中。

尽管这还是没有解释我为什么使用make_pair。make_pair是一个帮助函数模板，它能够用你给定的参数来创建一个pair对象。一些人更愿意使用这个函数模板而不愿意调用这个pair类的构造函数，这是因为一个类模板不能使用参数推演来计算出它的模板参数，而函数模板就可以。因此，下面两个代码在功能上是相同的。

```
strMap.insert(std::make_pair("Sunday", "Sonntag"));
strMap.insert(std::pair<string, string>("Sunday", "Sonntag"));
```

map不允许重复的关键字。如果你需要允许重复的关键字的话，你就需要使用multimap，它允许存在多个相等的关键字。它的接口与map的接口是相同的，但是它的成员函数的行为与map的则不同。表6-1列出了这些成员函数，这些成员函数只存在

www.TopSage.com

map和multimap的一个中，而不存在于另一个中，并且在这些共同的成员函数中解释了它们行为上的不同之处。map和multimap都有一些typedef重定义，它们描述一些存储在map和multimap中的一些不同的值。在表6-1中，我如下这样使用它们：

key_type
: 这是关键字的类型。在一个声明为map<string, MyClass*>的字符串map中，key_type就是字符串类型。

mapped_type
: 这是关键字映射成的值的类型。在一个声明为map<string, MyClass*>的字符串map中，mapped_type的类型就是MyClass*。

value_type
: 这是包含关键字和值的对象的类型，在一个map或multimap中，它是pair<const key_type, mapped_type>。

表6-1 map与multimap

成员函数	map、multimap或者是两者都有	行为
T& operator[] (const key_type& k)	map	返回一个指向值对象的索引，这个值对象与关键字k一起存储。如果k不存在这个map中，并且这个关键字将会被追加到这个map中，并且使用值类型的默认构造函数来创建一个值对象
iterator insert(const value_type& v) pair<iterator, bool> insert(const value_type& v)	两个都有	第一个版本把v插入到这个multimap中并且返回一个指向这个插入的pair对象的迭代器。第二个版本如果在这个map中没有与v的关键字相等的关键字，就把v插入到这个map中。如果有，返回的pair对象包含一个指向这个已被插入的pair对象的迭代器，并且无论这个插入成功与否都会有一个布尔值返回来指明

www.TopSage.com

表6-1 map与multimap（续）

成员函数	map、multimap或者是两者都有	行为
iterator find(const key_type& k)	两者都有	返回一个迭代器或者一个const_iterator迭代器，它指向这个与k对应的mapped_type。在一个multimap中，这个返回的迭代器并不能保证指向等于k的第一个值。如果没有与k相等的关键字存在，那么返回的迭代器就等于end()

表6-1也说明了在map和multimap之间行为上的不同。

如果操作符[]不能满足你的要求，map还有别的方法可以用来查找元素。你可以使用find成员函数：

```
map<string, string>::const_iterator p
   = strMap.find("Thursday");

if (p != strMap.end())
    cout << "Thursday = " << p->second << endl;
```

仅需要注意到的是当你使用multimap时，这个返回的项目并不能保证是与你正要查找的关键字相等的第一个元素。如果你需要的是不比某个值小或不比某个值大的第一个元素，你就可以使用lower_bound或者upper_bound。lower_bound返回一个迭代器，它指向等于或大于这个key_type参数的第一个键/值对。换句话说，如果你的map填充的是一个星期的天数的话，如示例6-6那样，接下来就会返回一个指向这个包含“Friday”和“Freitag”键/值对的迭代器。

```
p = strMap.lower_bound("Foo");

if (p != strMap.end())
    cout << p->first << " = " << p->second << endl;
```

这是因为“Friday”是第一个大于或等于“Foo”的关键字。upper_bound的工作方式相同，但是方向相反。

就像在讨论的开始我说明的那样，map中存储的元素是按照它们关键字排序存储的，因此如果你从开始到末尾这样去枚举它们的话，那么每一元素都会比前面的元素大（在multimap中是它不会比它前面的元素小）。但是如果你使用的并不是字符串或者数字作为你的关键字的话，当这个map不得不决定什么应该被插入，在哪里插入时，你可能不得不指定这些关键字是如何比较的。

www.TopSage.com

默认的情况下，关键字都是使用标准的算符less（声明在<functional>中）。less是一个二元函数（带两个同样类型的参数）并且返回一个布尔值指明第一个参数是否比第二参数小。换句话说，less(a, b)返回a < b。如果这不是你想要的话，你就需要创建你自己的算符并且声明这个map使用它。例如，如果你需要一个Person对象作为你的关键字，并且每一个Person都有姓和名，你就需要比较姓和名。示例6-7说明了这样做的一种方法。

示例6-7 使用你自己的排序算符

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

class Person {
   friend class PersonLessThan;
public:
   Person(const string& first, const string& last) :
 lastName_(last), firstName_(first) {}
   // ...
   string getFirstName() const {return(firstName_);}
   string getLastName() const {return(lastName_);}
private:
   string lastName_;
   string firstName_;
};

class PersonLessThan {
public:
   bool operator()(const Person& per1,
    const Person& per2) const {
 if (per1.lastName_ < per2.lastName_)           // Compare last
    return(true);                               // names, then
 else if (per1.lastName_ == per2.lastName_)     // first
    return(per1.firstName_ < per2.firstName_);
 else
    return(false);
   }
};

int main() {

   map<Person, string, PersonLessThan> personMap;

   Person per1("Billy", "Silly"),
      per2("Johnny", "Goofball"),
      per3("Frank", "Stank"),
      per4("Albert", "Goofball");

   personMap[per1] = "cool";
   personMap[per2] = "not cool";
   personMap[per3] = "not cool";
   personMap[per4] = "cool";

   for (map<Person, string, PersonLessThan>::const_iterator p =
        personMap.begin(); p != personMap.end(); ++p) {
      cout << p->first.getFirstName() << " " << p->first.getLastName()
           << " is " << p->second << endl;
   }
}
```

www.TopSage.com

map是用来存储键/值对的最后的方法。一旦你理解了这些微妙的行为，如[]操作符是如何工作的和这个键/值对是如何存储的（作为<Key, Value>键/值对对象）的话，你就会觉得map很好使用并且具有很好的性能。

请参考6.7节。

6.7 使用哈希函数

问题

如何存储关键字和值，且需要在常数时间内访问对应的元素，但不希望元素是以排序的方式存储的？

解决方案

可以使用哈希相关的容器，如hash_map或者hash_set。但是，请注意，这些容器并不是C++标准指定的标准容器，而是大部分标准类库实现包含的扩展。示例6-8说明该如何使用一个hash_set。

示例6-8 在一个hasn_set中存储字符串

```
#include <iostream>
#include <string>
#include <hash_set>

int main() {

   hash_set<std::string> hsString;
   string s = "bravo";

   hsString.insert(s);
   s = "alpha";
   hsString.insert(s);
   s = "charlie";
   hsString.insert(s);

   for (hash_set<string>::const_iterator p = hsString.begin( );
        p != hsString.end(); ++p)
      cout << *p << endl; // Note that these aren't guaranteed
                          // to be in sorted order
}
```

讨论

哈希容器在任何语言中都是流行的数据结构，但是不幸的是，C++标准并没有要求提供关于它们的实现。但是，如果你使用一个哈希容器的话，你就会发现所有的实现都有哈希容器的实现：你使用的这些标准类库的实现都包含hash_map和hash_set，但是由于它们都不是标准要求的，导致它们在各个标准类库中实现的接口可能不同。我将描述在STLPort标准类库的实现中提供的哈希容器。

www.TopSage.com

大家网 TopSage.com

注意： STLPort是一个存在很长时间的免费的可移植的标准类库实现，并且它提供了哈希容器。如果你使用不同的类库，它们的接口可能不同，但总的思想是相同的。

哈希容器（在大部分C++社区中称之为哈希关联的容器）的主要特性是在平均的情况下，它给元素的定位、插入和删除提供了常数时间的操作代价；在最坏的情况下，操作也只需线性复杂度。为了这些操作的常数时间的折衷是在这个哈希容器中的元素不是按顺序存储的，因为它们存储在一个map中。

请看示例6-8。使用一个哈希容器（这个例子中，是一个hash_set）是很简单的，就像大部分别的容器一样声明它并开始在这个容器中插入元素：

```
hash_set<string> hsString;  // A hash_set of strings
string s = "bravo";
hsString.insert(s);   // Insert a copy of s
```

使用一个hash_map也是类似的，除了（最少）你不得不指定它们要使用的关键字和数据类型之外。这与map是相同的：

```
hash_map<string, string>
   hmStrings;  // Map strings to strings
string key = "key";
string val = "val";
hmStrings[key] = val;
```

对于使用哈希容器，这些都是基本的；还有一些有用的模板参数可以让你指定使用的哈希函数，这个函数是用来测试关键字是否相等的，同时也可以指定一个内存分配的对象。稍后我将讨论它们。

在大部分类库中都有四个哈希容器，并且它们组成了这个标准类库的其他的关联容器（例如，map和set），它们是hash_map、hash_multimap、hash_set和hash_multiset。哈希容器都是使用一个哈希表来实现的。一个哈希表是一个数据结构，基本上是通过一个哈希函数使得元素的访问能在常数时间内完成，这个哈希函数能够跳到这个期望的元素存储的位置最接近的位置，而不是在一个类似于树结构中反复转换。

STLPort中基于哈希表的容器声明如下所示：

```
hash_map<Key,                    // The type of the key
    Value,                       // The type of the value
                                 // associated with the key
    HashFun = hash<Key>,         // The hash function to use
    EqualKey = equal_to<Key>,    // Function to use for key
                                 // equivalence test
    Alloc = alloc>               // The allocator to use

hash_set<Key,                    // The type of the key
    HashFun = hash<Key>,         // The hash function to use
    EqualKey = equal_to<Key>,    // Function to use for key
                                 // equivalence test
    Alloc = alloc>               // The allocator to use
```

www.TopSage.com

一个hash_map是一个存储的<const Key, Data>。意思是说当我在下面描述哈希表时，表中的元素是键/值对；hash_map并不是分开存储键和值（map也不是）。一个hash_set则是把键作为值类型存储。

这个HashFun模板参数是一个函数，它返回一个键到值类型的对象，这个值可以被作为size_t类型存储。下面将会讨论更多。这个EqualKey模板参数是一个函数，它带两个参数并且如果它们相等的话，就返回真值。在hash_map和hash_set容器中，没有两个关键字是可以相等的；而hash_multimap和hash_multiset则可以有两个相等的关键字。就像所有的容器一样，Alloc都是将要被使用的内存分配器。

一个哈希表有两部分。有一个相对比较大的矢量，它的每一个索引称为一个桶。每一个桶事实上是一个指针，它指向一个相对短一些的单向链表或双向链表（在STLPort中是单向链表），这些链表是事实上的数据存储的地方。你可以通过这个bucket_count成员函数得到一个哈希容器中桶的个数。图6-3说明了一个哈希map在内存中的映像。

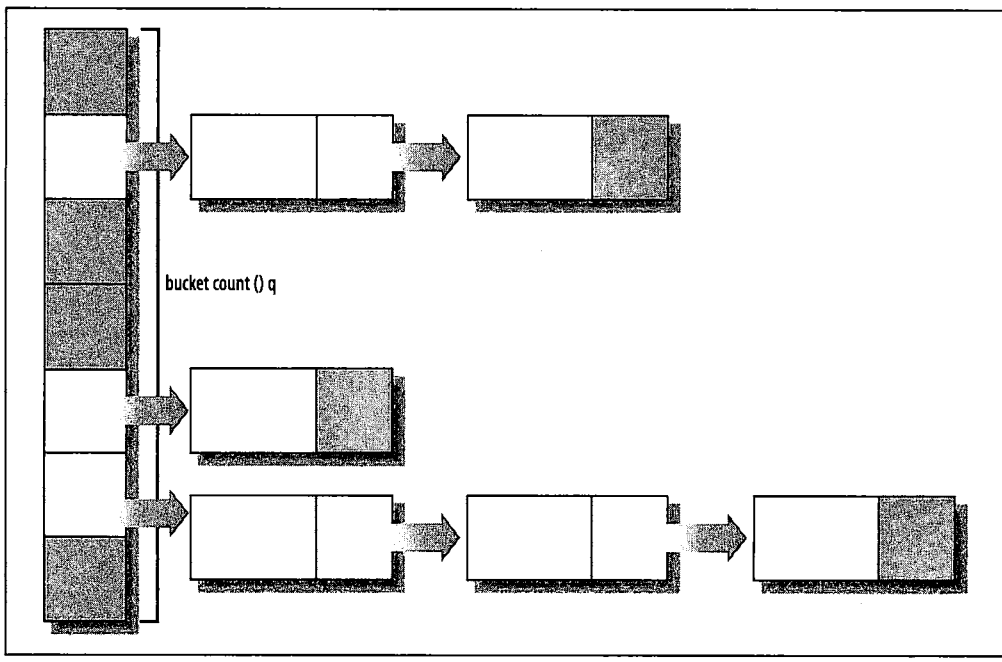


图6-3 一个哈希表

考虑示例6-8中hash_set的使用。当你插入一个元素时，这个容器首先不得不计算出这个元素属于哪个桶。它通过传入关键字来调用这个容器的哈希函数来实现这个目的（哈希函数一会再讨论）并且使用这个桶的数目来计算出它的模。这就可以给出在这个桶矢量中的索引。

www.TopSage.com

如果你对“哈希”这个词不是很熟悉的话，它是一个直接的概念。假设给定一个值（例如，一个字符数组），一个哈希函数就是一个带一个简单的参数并且返回一个size_t类型的哈希值（例如是一个数字）。理想的情况下，你需要一个哈希函数，它产生的哈希值常常是唯一的，但是不必总是唯一的。这个函数不是数学意义上的一一对应的关系：多个字符串可以对应同一个哈希值。一会后我将讨论它为什么可以这样。

STLPort包含一个哈希函数如在<hash_map>和<hash_set>中的函数模板一样。这个函数不能为任何对象工作，因为不可能给所有的类型都做一个泛型哈希函数并且这个哈希函数还能为任何类型的输入工作正常。相反，对于一些内置的类型有一些特殊的处理，这些类型经常在一个哈希表中作为关键字使用。例如，如果你想看一个哈希值到底是什么样子的，你就可以哈希一个字符串：

```
std::hash<const char*> hashFun;

std::cout << "\"Hashomatic\" hashes to "
<< hashFun("Hashomatic") << '\n';
```

你看到如下所示：

```
"Hashomatic" hashes to 189555649
```

STLPort给这些类型提供了一些特殊化处理：char*、const char*、char、unsigned char、signed char、short、unsigned short、int、unsigned int、long和unsigned long。这看起来很多，但最终它的意思是这个类库给字符串和数字类型提供了一些内在的哈希函数支持。如果你需要哈希别的内型，你就需要提供你自己的哈希函数。

当你需要在一个哈希表中加入元素时，它就使用这个模操作和桶的个数来计算出这个元素属于哪个桶，如hashFun(key) % bucket_count()。这是一个非常快的操作，它指向了这个桶在这个主矢量中正确的索引。

你可以如同任何普通的联合容器那样使用一个哈希容器，例如使用操作符[]来在一个hash_map中追加元素。不同之处就是你需要知道插入和查找的时间是常数时间而不是对数级时间。考虑一下示例6-9，它包含了一个简单的类，这个类把用户的注册名映射成会话对象。它使用了这节中我讨论的有关哈希容器的一些特性。

示例6-9 一个简单的会话管理器

```
#include <iostream>
#include <string>
#include <hash_map>

using namespace std;

class Session { /* ... */ };

// Make reading easier with a typedef
typedef hash_map<string, Session*> SessionHashMap;
```

www.TopSage.com

```
class SessionManager {

public:
   SessionManager () : sessionMap_(500) {}  // Initialize hash table
                                            // with 500 buckets
  ~SessionManager () {
     for (SessionHashMap::iterator p = sessionMap_.begin();
          p != sessionMap_.end(  ); ++p)
        delete (*p).second; // Destroy the Session object
   }

   Session* addSession(const string& login) {
 Session* p = NULL;
 if (!(p = getSession(login))) {
    p = new Session();
    sessionMap_[login] = p; // Assign the new session with
 }                          // operator[]
 return(p);
   }
   Session* getSession(const string& login) {
     return(sessionMap_[login]);
   }
   // ...

private:
   SessionHashMap sessionMap_;
};
```

每个关键字都映射成一个单桶，并且在一个桶中可以有多个关键字。一个桶常常是一个单向或双向链表。

关于哈希函数和哈希表有非常多的文献资料。如果你对这类文献感兴趣，你可以在Google上查找“C++ hash function”。

请参考6.6节。

6.8 按顺序存储对象

问题

如何按顺序存储对象？因为你需要经常访问这些排序的对象并且不想每次访问的时候对它们重新排序。

解决方案

使用这个联合的容器集合，它声明在<set>中，按序存储元素。它使用标准的小于类模板（它将在它的参数上调用<操作符），默认的情况下，或者你也可以提供你自己的排序断言。示例6-10说明了该如何在一个集合中存储字符串。

示例6-10 在一个集合中存储字符串

```
#include <iostream>
#include <set>
#include <string>

using namespace std;
```

www.TopSage.com

```
int main() {

   set<string> setStr;
   string s = "Bill";

   setStr.insert(s);
   s = "Steve";
   setStr.insert(s);
   s = "Randy";
   setStr.insert(s);
   s = "Howard";
   setStr.insert(s);

   for (set<string>::const_iterator p = setStr.begin(  );
        p != setStr.end(  ); ++p)
      cout << *p << endl;
}
```

由于存储的值是已排序的，所以它的输出如下所示：

```
Bill
Howard
Randy
Steve
```

讨论

一个集合是一个联合容器，在插入和查找操作上的复杂度都是对数级的，而删除操作则是常数时间（一旦你找到这个元素的话，你就可以删除它）。集合是唯一的联合容器，也就意味着在它里面的元素都是不相等的，尽管如果你需要存储相等的多个元素的话，你也可使用一个multiset。你可以把这个集合想象成数学意义上的集合，这就是说，元素的集合带有追加操作的代价，这个代价是为了维持这些元素的顺序的。

你可以插入或查找元素，但是，就像一个list一样，一个集合是不允许随机访问元素的。如果你需要访问这个集合中的某个元素，你需要使用查找成员函数来找到它，或者使用set<T>::iterator或set<T>::const_iterator迭代器来枚举这些元素。

一个集合的声明如下：

```
set<typename Key, // The type of the element
    typename LessThanFun = std::less<Key>, // The function/functor
                                           // used for sorting
    typename Alloc = std::allocator<Key> > // Memory allocator
```

你需要总是指定这个关键字Key，有时你也需要提供你自己的LessThanFun比较函数，并且你总是不需要提供你自己的分配函数allocator（因此这里就不讨论如何实现一个分配函数allocator）。

这个关键字Key模板参数就像在很多其他的标准容器中的那样，这个元素的类型将要被存储。它是在集合上的set<Key>::key_type的typedef重定义，因此你可以在运行时访问这个类型。这个关键字Key类不得不支持复制构造和赋值操作。

www.TopSage.com

大学网 TopSage.com

示例6-10说明了该如何使用一个字符集合。使用一个集合来存储任何其他类型的对象也是同样的工作原理；使用模板参数作为类名来声明这个集合：

```
std::set<MyClass> setMyObjs;
```

使用一种可能最简单的方法来使用一个集合，这就是你所有你需要做的。但是大多数情况下，生活不会是这样的简单。例如，如果你需要在一个集合中存储指针的话，你就不能依赖于默认的排序断言，因为这个默认的排序只是使用它们的地址来排序。为了使得元素能正确地排序，你需要为小于操作提供你自己的断言。示例6-11说明该如何这样做。

示例6-11 在一个集合中存储指针

#include <iostream>

```
#include <set>
#include <string>
#include <functional>
#include <cassert>

using namespace std;

// Class for comparing strings given two string pointers
struct strPtrLess {
   bool operator()(const string* p1,
    const string* p2) {
   assert(p1 && p2);
   return(*p1 < *p2);
   }
};

int main() {
   set<string*, strPtrLess> setStrPtr;  // Give it my special
                                        // less-than functor
   string s1 = "Tom";
   string s2 = "Dick";
   string s3 = "Harry";

   setStrPtr.insert(&s1);
   setStrPtr.insert(&s2);
   setStrPtr.insert(&s3);

   for (set<string*, strPtrLess>::const_iterator p =
   setStrPtr.begin(); p != setStrPtr.end(); ++p)
 cout << **p << endl;  // Dereference the iterator and what
                       // it points to
}
```

如果由p1指向的字符串小于p2指向的字符串的话，strPtrLess返回真。这就使得它成为了一个二元的断言，因为它带两个参数并且返回一个布尔值。由于操作符<是给字符串定义的，我仅可以为了比较而使用它。事实上，如果你需要带一个更多的泛型方法，给你的比较断言使用类模板：

```
template<typename T>
class ptrLess {
public:
   bool operator()(const T* p1,
                   const T* p2) {
```

www.TopSage.com

```
        assert(p1 && p2);
        return(*p1 < *p2);
        }
    };
```

这就使得它能为任何带有定义过操作符<的任何指针工作。你可以如下一样声明一个集合：

```
set<string*, ptrLess<string> > setStrPtr;
```

集合也能支持许多标准序列容器中的函数（例如，begin、end、size、max_size），也支持其他的联合容器中的函数（如insert、erase、clear、find）。

当你正使用一个集合时，记住当你每一次修改集合的状态时，你都需要执行排序操作。当元素的个数很大时，追加和删除的对数复杂度可以增长得很快，你是否需要每次都排序？如果不是的话，把元素存储在一个矢量或链表中并且只是在需要的时候才给它们排序，这样能获得更好的性能，它们的复杂度常常是$n*\log(n)$数量级。

6.9 容器存储

问题

如何实现一个标准容器（列表、集合等），它有很多实例，并且你需要通过把它们存储到另一个容器中跟踪它们？

解决方案

把你的容器的指针存储到一个单一的主容器中。例如，你可以使用一个map来存储字符串的关键字和一个指向它的值的集合的指针。示例6-12说明了一个简单的事务日志类，它把它的数据字符串集合指针对存储到一个map中。

示例6-12 在一个map中存储集合指针

```
#include <iostream>
#include <set>
#include <map>
#include <string>

using namespace std;

typedef set<string> SetStr;
typedef map<string, SetStr*> MapStrSetStr;

// Dummy database class
class DBConn {
public:
   void beginTxn() {}
   void endTxn() {}
   void execSql(string& sql) {}
};

class SimpleTxnLog {
```

www.TopSage.com

```
public:
   SimpleTxnLog() {}
  ~SimpleTxnLog() {purge();}

   // Add an SQL statement to the list
   void addTxn(const string& id,
const string& sql) {
 SetStr* pSet = log_[id];      // This creates the entry for
 if (pSet == NULL) {           // this id if it isn't there
    pSet = new SetStr();
    log_[id] = pSet;
 }
 pSet->insert(sql);
   }

   // Apply the SQL statements to the database, one transaction
   // at a time
   void apply() {
 for (MapStrSetStr::iterator p = log_.begin(  );
 p != log_.end(); ++p) {
    conn_->beginTxn();

    // Remember that a map iterator actually refers to an object
    // of pair<Key,Val>.  The set pointer is stored in p->second.
    for (SetStr::iterator pSql = p->second->begin();
         pSql != p->second->end(); ++pSql) {
       string s = *pSql;
       conn_->execSql(s);
       cout << "Executing SQL: " << s << endl;
    }

    conn_->endTxn();
    delete p->second;
 }
 log_.clear();
   }

   void purge() {
      for (MapStrSetStr::iterator p = log_.begin();
           p != log_.end(  ); ++p)
         delete p->second;

      log_.clear();
   }
  // ...

private:
   MapStrSetStr log_;
   DBConn* conn_;
};
```

讨论

示例6-12提供了一种情况，在这种情况下，你可以把很多容器存储到一个容器中。假设你需要把一个SQL语句序列批量存储起来，在将来的某个时候在一个关系型数据库中执行。这就是SimpleTxnLog所做的。为了使得它更有用，可能还需要更多的成员函数，并且为了使得它更安全，还需要一些异常处理，但是这个例子的目的是为了说明该如何把一个类型的容器存储到另一个容器中去。

作为开始，为了使得代码更容易读，我创建了一些typedef重定义：

www.TopSage.com

```
typedef std::set<std::string> SetStr;
typedef std::map<std::string, SetStr*> MapStrSetStr;
```

当你使用模板时，它的声明将变得非常长，这将使得它很难读，因此为了使得更容易读，就使用了typedef。更进一步，使用typedef使得改变一些关于这个模板的声明更容易，这样就不用为了改变而在很多源文件中查找和替换。

这个DBConn类是一个虚拟类，它用来代替一个关系数据库的连接。当你接触到这个SimpleTxnLog的定义时，你就会发现一个非常有趣的部分是在这个addTxn成员函数中，在这个函数的开始处，我这样做的目的是为了查看这个传入的id对应的集合对象是否已经存在：

```
SetStr* pSet = log_[id];
```

log_是一个map（请看6.6节），因此操作符[]为id做了对应的查找，查找是否存在有与它相关的数据对象。如果存在的话，这个数据对象就被返回并且pSet将是非空的；如果不存在的话，它就创建它并且返回一个相关的指针，这个指针将是NULL。然后，我就可以验证这个pSet是否指向任何东西从而决定我是否需要创建另外一个集合：

```
if (pSet == NULL) {
   pSet = new SetStr();  // SetStr = std::set<std::string>
   log_[id] = pSet;
}
```

一旦创建这个集合，我就不得不把它赋给在这个map中一个相关的关键字，由于pSet是存在于这个map中的数据对象的一个副本（一个集合指针）而不是它的值本身。一旦我这样做的话，所剩下的就是把这个元素加到这个集合中并返回：

```
pSet->insert(sql);
```

通过上面的步骤，我就把指向一个容器（一个集合）地址的指针加到另一个容器（一个map）。我没有做的就是把一个集合对象加到一个map。这个区别是重要的。由于容器具有复制进和复制出语意，下面的这个动作将把整个集合复制进这个map中：

```
set<string> s;
// Load up s with data...
log_[id] = s; // Copy s and add the copy to log_
```

这将导致很多额外的复制动作，这是你不需要的。因此，当你使用容器的容器时，应该遵循的一般规则是使用容器指针的容器。

www.TopSage.com

第7章

算法

7.0 概述

这一章将要描述如何使用标准算法以及如何在标准容器上使用这些算法。这些算法最初是标准模板类库（STL）的一部分，标准模板类库是算法、迭代器和容器类这些现在属于标准类库的模板类的集合（第6章包含了一些标准容器的说明）。我将把它们简单地称为标准算法、迭代器和容器，但是请记住它们和别的一些作者称之为STL一部分是相同的。在这个标准类库中最重要的一个就是迭代器，并且在7.1节中将首先解释它们是什么和如何使用。以后的那些章节将介绍如何使用和扩展这些标准算法。最后，如果这个标准类库还不能满足你的需求，7.10节将告诉你如何写出你自己的算法。

有两个原因使得我在这些章节中主要来讨论标准容器上的用法。第一，标准容器是使用非常广泛的，最好先学会它而不是自己去重新发明。第二，标准类库中这些算法的实现在互操作和性能上都提供了一个很好的值得学习的模型。如果你看过标准类库中pros的代码，你都可能学到一些非常有价值的东西。

所有的标准算法都使用迭代器。即使你已经熟悉7.1节中迭代器的概念，你还是应该看一下表7-1，它包含一系列的惯例，在以后的章节中当我给这些标准算法列出它们的声明时我会用到。

表7-1 迭代器类型缩写

缩写	含义
In	输入迭代器
Out	输出迭代器
Fwd	向前迭代器
Bid	双向迭代器
Rand	随机访问迭代器

www.TopSage.com

大家网 TopSage.com

标准算法使用了函数对象或者算符。一个函数对象就是一个重载了操作符()的类，这个()括号操作符可以像函数一样被调用。一个返回一个布尔值（并且不包含状态，因此也可叫做纯的）的算符称做一个断言，它们是标准算法的另外一个固定的属性。总的来说，一个断言带一个或两个参数：如果它带一个参数，它就叫一元断言；如果它带两个参数，就叫二元断言。为了简单，当我列举函数声明时我就使用表 7-2 中的这些缩写。

表 7-2 算符类型

类型名	描述
UnPred	一个一元断言。带一个参数并返回一个布尔值
BinPred	一个二元断言。带两个参数并返回一个布尔值
UnFunc	一个一元函数。带一个参数并返回任意值
BinFunc	一个二元函数。带两个参数并返回任意值。

在大多数情况下，一个函数指针能被用作为一个算符。当我用算符这个术语时，在没有特殊声明的情况下，我就是指函数指针。

7.1 遍历容器

问题

有一批迭代器，它们大部分最可能来自于一个标准容器，并且这些标准算法又不满足你的需求，如何遍历这个容器？

解决方案

使用一个迭代器或常量迭代器来访问和向前遍历容器中的每个元素。在标准库中，算法和容器的通信是通过迭代器的，标准算法中一个恰当的想法就是迭代器的使用对你是透明的，除非是你实现自己的算法才需要你直接使用迭代器。即使如此，你还是应该理解各种迭代器的不同之处，你才能有效地使用标准的算法和容器。示例7-1中说明了迭代器的一些直接的用法。

示例 7-1 和容器一起使用迭代器

```
#include <iostream>
#include <list>
#include <algorithm>
#include <string>

using namespace std;

static const int ARRAY_SIZE = 5;

template<typename T,
         typename FwdIter>
FwdIter fixOutliersUBound(FwdIter p1,
```

www.TopSage.com

```
                          FwdIter p2,
                          const T& oldVal,
                          const T& newVal) {
   for ( ;p1 != p2; ++p1) {
      if (greater<T>(*p1, oldVal)) {
         *p1 = newVal;
      }
   }
}

int main() {

   list<string> lstStr;

   lstStr.push_back("Please");
   lstStr.push_back("leave");
   lstStr.push_back("a");
   lstStr.push_back("message");

   // Create an iterator for stepping through the list
   for (list<string>::iterator p = lstStr.begin(  );
        p != lstStr.end(  ); ++p) {
      cout << *p << endl;
   }

   // Or I can use a reverse_iterator to go from the end
   // to the beginning.  rbegin returns a reverse_iterator
   // to the last element and rend returns a reverse_iterator
   // to one-before-the-first.
   for (list<string>::reverse_iterator p = lstStr.rbegin(  );
        p != lstStr.rend(  ); ++p) {
      cout << *p << endl;
   }

   // Iterating through a range
   string arrStr[ARRAY_SIZE] = {"My", "cup", "cup", "runneth", "over"};

   for (string* p = &arrStr[0];
        p != &arrStr[ARRAY_SIZE]; ++p) {
      cout << *p << endl;
   }

   // Using standard algorithms with a standard sequence
   list<string> lstStrDest;
   unique_copy(&arrStr[0], &arrStr[ARRAY_SIZE],
                     back_inserter(lstStrDest));
}
```

讨论

一个迭代器是一个用来指向容器中的一个简单对象的类型。标准容器中，访问它包含元素的主要手段就是迭代器。一个迭代器的行为就像是一个指针：为了访问一个迭代器所指的对象，你可以使用星号（*）或（->）来间接引用一个迭代器，并且你可以用看上去与指针的算术运算相同的方式来向前和向后移动一个迭代器。然而，在很多方面，迭代器和指针还是不同的。在探讨这些之前，让我们先来讨论如何使用迭代器这个最基本的技术。

使用迭代器

你可以使用一个你将要遍历的类型来声明一个迭代器。例如，在示例7-1中，我使用`list<string>`类型，因此我就像如下这一行一样声明一个迭代器：

www.TopSage.com

```
list<string>::iterator p = lstStr.begin();
```

如果你不经常使用标准容器的话，也许声明中::interator部分会看上去有点奇怪。它是list类模板的一个嵌套的typedef定义，以便于容器的用户创建一个模板的某个特殊实例的迭代器。这是所有的标准容器都遵守的一个标准惯例；例如，你可以像如下代码一样声明一个list<int>类型或set<Myclass>类型的迭代器：

```
list<int>::iterator p1;
set<MyClass>::iterator p2;
```

让我们再回到前面那个例子，迭代器p被初始化指向序列中的第一个元素，这个元素是由begin函数返回的。为了向前移动指向下一个元素，你可以使用++自增操作符。你可以使用前自增（++p）或后自增（p++），就像你用一个指针指向一个数组元素一样，但前自增不会每次都创建一个返回的临时值，因此它的效率更高，也推荐使用它。后自增（p++）则需要创建一个临时变量因为它要返回一个p增加之前的值。然而，在它返回以后，它不可能增加它的值，因此它就不得不需要创建当前值的一个副本，并增加这个当前值，然后再返回这个临时值。也即是创建一个临时变量然后再增加它本身的值，因此如果你不需要后自增这种行为的话，就使用前自增。

当你碰到序列的末端时，你就知道你应该停止向前移动到下一个元素。或者更严格地说，当你碰到序列末端之后一个元素时。标准容器的惯例是用一个神秘的值来代表序列末端之后的那个元素，并且这个元素由end函数返回。就像例子中那样，在一个for循环中，一个便利的工作如下所示：

```
for (list<string>::iterator p = lstStr.begin();
     p != lstStr.end(); ++p) {
   cout << *p << endl;
}
```

只要p等于end函数返回的值，p就不可能再向前移动了。如果容器是空的，那么begin和end就是相等的，因此循环就不会被执行（然而，测试一个容器是否是空的还是应使用empty成员函数，不能使用begin与end相比较或检测size == 0是否成立）。

这是迭代器的一个简单的函数解释，但不全是。首先，就像在前面声明的那样一个迭代器可以作为右值或左值来工作，这就是说你可以给它赋一个迭代器引用的对象或用迭代器赋给它。为了重写字符串链表中的每个元素，我可以如下这样写：

```
for (list<string>::iterator p = lstStr.begin();
     p != lstStr.end(); ++p) {
   *p = "mustard";
}
```

由于*p引用一个字符串类型的对象，string::operator=(const char*)被用来给容器中的一个元素赋一个新值。但如果lstStr是一个常量对象那该如何是好？在这个例子中，这个迭代器将不能正常工作，因为它返回一个非常量对象。你可以使用const_iterator迭代器，它是一个只返回右值的迭代器。假设你决定要写一个打印容器中的内容的简单函数，自然地，你可想把这个容器作为常量引用传给这个函数：

www.TopSage.com

```
template<typename T>
void printElements(const T& cont) {
   for(T::const_iterator p = cont.begin();
       p != cont.end(); ++p) {
      cout << *p << endl;
   }
}
```

在这种情况下，使用const是非常对的，并且如果你的代码修改*p的值，这个const_iterator迭代器将使得编译器能给你报错。

你可能需要做的另一件事是你想向后遍历容器。你同样可以使用一个平常的迭代器，但为这个目的，标准库中有一个reverse_iterator的迭代器。一个reverse_iterator迭代器看上去和一个平常的迭代器一样，但它增加和减小的方式刚好和平常的迭代器相反，并且为了代替使用平常迭代器的begin和end方法，你可以使用rbegin和rend方法，它们返回reverse_iterator迭代器。一个reverse_iterator迭代器用一个相反的方式看待容器。例如，不要用begin方法初始化一个reverse_iterator迭代器，你可以使用rbegin方法，它返回一个reverse_iterator迭代器指向序列中的最后一个元素。自增操作符++向后移动或者说它是朝着序列头部移动，而rend函数返回一个指向序列中第一个元素前的一个。下面是一个例子：

```
for (list<string>::reverse_iterator p = lstStr.rbegin();
     p != lstStr.rend(); ++p) {
   cout << *p << endl;
}
```

但也许你不能使用reverse_iterator迭代器，在这种情况下，你可以与下面一样使用平常的迭代器：

```
for (list<string>::iterator p = --lstStr.end(  );
     p != --lstStr.begin(); --p) {
   cout << *p << endl;
}
```

最后，如果你碰巧知道向前或向后移动多少，你就可以用一个整数来做运算从而向前或向后移动多个位置。如果你想移动到这个链表的中间位置，你就可以如下一样做：

```
size_t i = lstStr.size();

list<string>::iterator p = begin();
p += i/2;  // Move to the middle of the sequence
```

但是请注意：这个操作的代价可能是常数或是线性复杂度，这依赖于容器的类型。你如果使用的是那种连续存储元素的容器，如矢量或双端队列，那么这个迭代器就能计算出它应该跳转到哪并花费一个常数的时间。但如果你使用的是一个基于节点的容器，如链表，你就不能使用随机访问的操作。并且你就不得不遍历直到找到你需要的那个元素。这是非常昂贵的。这就是为什么需要根据你如何遍历容器或者如何查找元素这些实际的情况来选定最好的容器的原因（更多的关于容器是如何工作的信息请参考第6章）。

www.TopSage.com

当你使用的容器允许随机访问时，你就应该使用迭代器而不是使用一个索引下标来使用操作符[]访问每个元素。如果你正在写一个作为函数模板的通用算法时，这是非常重要的。因为并不是所有的容器都支持随机访问。

迭代器还可以有一些别的用法，但不是任意的迭代器都有的功能。根据功能上的不同，迭代器可以分为五类。它们不是与类的继承那么简单，以下就是这些类型的描述。

迭代器类型

不同种类的容器提供的迭代器并不一定要完成相同的功能。例如。一个vector<T>::iterator迭代器可以使用+=操作符向前跳过几个元素，而list<T>::iterator迭代器则不能。这就是这两个迭代器在类型上的不同。

一个迭代器的类型从概念上来说是一个接口（不是从技术来说；并没有使用一个抽象基类来实现迭代器类型）。总共有五个类型，每一个类型都提供了一个递增的功能集。下面就是这五个类型的描述，按功能从少到多排列：

输入迭代器

输入迭代器用p++或++p支持向前移动，通过*p间接引用支持迭代器取它指向的对象值。通过*p你可以取得右值。输入迭代器常常被一些像流一样的对象使用，间接引用一个输入迭代器就意味着把下一个元素从流中取出来，因此，你就可以用它只读那些特别元素。

输出迭代器

输出迭代器用p++或++p支持向前移动，通过*p间接引用支持迭代器取它指向的对象值。它和输入迭代器是不同的，因为你不能从它读入，你只能向它写入，并且一次只能写一个元素。不像输入迭代器，你可以得到一个左值而不是右值，因此你可以向它赋值但不可从它读。

向前迭代器

向前迭代器合并了输入迭代器和输出迭代器的功能：它支持++p和p++，并且你可以把*p当成是右值或左值。你可以在任何需要输入或输出迭代器的地方使用向前迭代器，这样你就可以任意地从间接引用向前迭代器中读或向它写入。

双向迭代器

就像它的名字一样，一个双向迭代器支持向前和向后移动。它是一个增加了--p或p--的向后移动功能的向前迭代器。

随机访问迭代器

一个随机访问迭代器支持双向迭代器的所有功能，同时还支持类似指针的操作。你可以使用p*[n]*来访问序列中p之后n个位置后的元素，同时你也能通过使用+、+=、-或者-=等减或加操作来访问前面某个位置的元素，并且这个操作只花费常

www.TopSage.com

大家网 TopSage.com

数时间。你也能通过使用 <、<=、> 或 >= 来比较两个迭代器 p1 和 p2 来决定相互的顺序（只要它们都指向同一个序列）。

或者你更愿意通过维恩图来查看这些类型的关系。请看图 7-1。

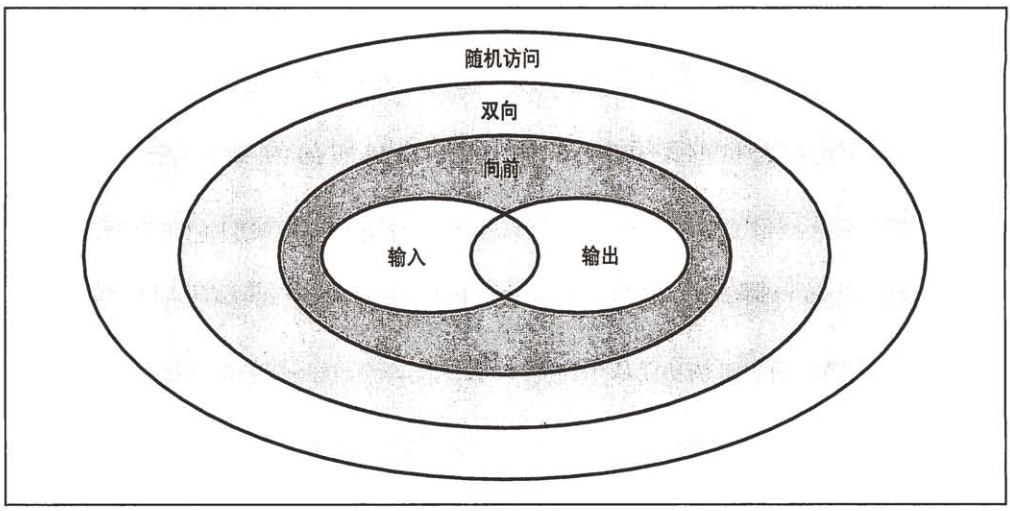


图 7-1 迭代器类型

大部分容器最少都支持双向迭代器，一些（矢量和双端队列）还提供了随机访问迭代器。一个容器支持的迭代器种类在标准中都会指明的。

大部分时间你都是为一个简单的任务使用迭代器：找到一个元素并删除它，或者做一些别的操作。为这种目的，你只需要一个向前迭代器，这是所有的容器都提供的。但当你需要写一个非平凡算法或使用一个来自标准类库的算法时，你常常需要的不只是一个简单的向前迭代器。这个时候，你该如何指明呢？这就要用到迭代器类型。

不同类型的迭代器让标准（或非标准）算法指明它们需要的功能的范围。总的来说，与迭代器一起使用的标准算法操作在某个范围的容器里而不是所有的容器。标准算法的声明会告诉你它们需要什么类型的迭代器。例如，std::sort 需要一个随机访问迭代器，因为它需要在一个常数的时间里引用一个非连续的元素。因此，这个排序函数的声明如下：

```
template<typename RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

通过这个迭代器类型的名字，你就能猜出它需要一个随机访问迭代器。你如果使用一个别的非随机访问迭代器的话，编译器将会报错，因为没有别的迭代器类型实现了类似指针的算术操作。

www.TopSage.com

一个特定容器提供的迭代器类型和某个特定标准算法需要的迭代器类型决定了这个算法能跟什么样的容器一起工作。这章的其他各节将要讨论各种标准算法。表7-1中列出了这些我在这章的其他部分使用每种算法能作为参数接受的迭代器。

这一节讨论了这些能跟容器一起使用的迭代器。迭代器的模式并不是特定于容器的，因此还有一些其他类型的迭代器。如流迭代器、流缓冲迭代器和原始数据存储迭代器等，但这里都没有讨论。

请参考第6章。

7.2 删除对象

问题

如何从一个容器中把某个元素删除掉？

解决方案

你可以使用容器的erase成员函数把某一个或某个范围内的元素删除，并且有可能使用标准算法中的一个算法使这个删除工作更容易。示例7-2给出几个不同的方法从一个序列中删除元素。

示例7-2 从容器中删除元素

```
#include <iostream>
#include <string>
#include <list>
#include <algorithm>
#include <functional>
#include "utils.h" // For printContainer(): see 7.10

using namespace std;

int main() {

   list<string> lstStr;

   lstStr.push_back("On");
   lstStr.push_back("a");
   lstStr.push_back("cloudy");
   lstStr.push_back("cloudy");
   lstStr.push_back("day");

   list<string>::iterator p;

   // Find what you want with find
   p = find(lstStr.begin(), lstStr.end(), "day");

   p = lstStr.erase(p); // Now p points to the last element

   // Or, to erase all occurrences of something, use remove
   lstStr.erase(remove(lstStr.begin(), lstStr.end(), "cloudy"),
                lstStr.end());
   printContainer(lstStr); // See 7.10
}
```

www.TopSage.com

讨论

使用容器的erase成员函数来删除一个或多个元素。所有的容器都有两个重载的erase成员函数：一个带一个指向你要删除的元素的简单迭代器作为参数，另一个则带两个迭代器，它们指向你要删除元素的范围。为了删除一个简单的元素，首先你需要获得指向这个元素的迭代器然后再把这个迭代器传给 erase 成员函数，如示例 7-2 中那样：

```
p = find(lstStr.begin(), lstStr.end(), "day");
p = lstStr.erase(p);
```

通过调用 p 指向元素的析构函数来把 p 指向的对象删除，并且给容器中余下的元素做一些重组工作。这个余下元素的重组工作依赖于容器的类型，因此这个操作的复杂度将因为容器的不同而不同。这个函数在序列容器和集合容器上函数的签名和行为上都有一些不同。

在序列容器中，erase 函数将返回一个迭代器，这个迭代器指向那个仅跟在最后一个被删除元素后的那个元素，如果序列中最后一个元素是最后被删除的话，这个迭代器就指向 end。这个操作的复杂度也会由于序列的不同而不同，因为序列也可以用很多不同的方式实现。例如，由于所有在矢量中的元素都存储在一个连续的内存中，为了填补被删除元素留下的空白，除了是在序列的末端之外的任何位置删除元素，那些跟在这个被删除元素后的所有元素都需要向序列的头部移动。这对性能的影响是很大的（线性），这就是如果你需要在序列中除了序列的末端之外经常删除（或插入等操作）元素的话，不应该选择使用矢量的原因。在 6.2 节中就这个问题有更详细的讨论。

在集合容器中，erase 成员函数返回 void 类型。如果你只是删除一个简单的元素，它的复杂度是一个常量，如果你需要删除一个范围内的元素，它的复杂度就是对数再加上删除元素的个数。这是因为集合容器常常用一个平衡树（如红黑树）来实现。

删除元素是容易的，但不是很有趣。你如果想在如何表达什么应该被删除上有更多的灵活性，你就需要参考标准算法（在 <algorithm> 文件中）。考虑来自示例 7-2 中的如下一行代码：

```
lstStr.erase(std::remove(lstStr.begin(), lstStr.end(), "cloudy"),
             lstStr.end());
```

请注意，我然仍使用了 erase 成员函数，但这次我想把我的 list<string> 中的所有的“cloudy”都删除掉。remove函数返回一个迭代器，我把这个返回的迭代器传给erase函数作为要删除范围的开始，并且我再把end函数传给 erase 函数作为要删除范围的结尾。它将删除这个范围中所有等于“cloudy”对象（调用它的 delete 函数）。但也许它的行为有点和你希望的不同。下面我将澄清一些术语。

事实上，remove 函数并不删除任何东西。它把那些不等于你指定的值的元素移动到序列的开始处，并且返回一个指向那些仅跟在这些元素后的那个元素的迭代器。然后，由

www.TopSage.com

你决定调用erase函数去删除在[p, end)之间的这些元素，这里p就是由remove函数返回的那个迭代器。

remove函数也有一些变体。你如果想删除那些满足某个断言而不是简单地等于某些值的元素，你该如何做呢？你可以使用remove_if函数。例如，假设你有一个名为Conn的类，它代表一些连接。如果某个连接的空闲时间比某个给定的值大的话，你就需要删除这个连接。首先，你就可以和下面一样创建一个函数：

```
struct IdleConnFn :
   public std::unary_function<const Conn, bool> {  // Include this line
   bool operator() (const Conn& c) const {         // so it works with
 if (c.getIdleTime() > TIMEOUT) {                  // other stuff in
   return(true);                                   // <functional>
 }
 else
   return(false);
   }
} idle;
```

然后，你就可以把你的函数传给remove_if函数并调用erase函数，就像下面一样：

```
vec.erase(std::remove_if(vec.begin(), vec.end(), idle), vec.end());
```

如果你为一个好的原因想从unary_function函数派生出别的函数的话。unary_function函数定义一些能被<functional>文件中其他函数使用的typedef，并且如果它们不存在的话，这些来自<functional>的函数将不能被编译。例如，如果你特别恶意的话，并且你想删除所有的不是空闲的连接，你就可以在你的空闲检测函数前加上not1，就像下面这样：

```
vec.erase(std::remove_if(vec.begin(), vec.end(), std::not1(idle)),
          vec.end());
```

最后，你如果不想改变原来的序列（或许它是常量型）并且你又想把结果复制到一个新的序列中的话。你可以使用remove_copy和remove_copy_if函数，它们与remove和remove_if的工作方式相同，只是它们带一个输出迭代器作为参数用来存储删除后的结果。例如，复制一个删除“cloudy“后的序列到一个新的序列中：

```
std::remove_copy(lstStr.begin(), lstStr.end(), lstStr2, "cloudy");
```

当你使用remove_copy或者别的标准算法写一个输出范围时，你不得不记住的是输出的范围必须足够大，能容纳所有这些将要被输出的元素。

erase和remove函数（或别的与它们相关的函数）为从一个序列中删除元素提供了一个方便的方法。它们给你自己去循环查找你需要删除的元素，找到后一个一个删除，这种方法提供了一个干净的现成的替代方法。

请参考6.2节和7.1节。

www.TopSage.com

7.3 随机打乱数据

问题

有一系列的数据，如何随机地打乱这些数据。

解决方案

使用定义在<algorithm>中的random_shuffle标准算法你就可以达到这个目的。random_shuffle带两个随机访问迭代器，和（可选的）一个随机数产生函数，并且在一个范围内随机打乱这些数据，示例7-3给出了一个简单的例子。

示例7-3 随机打乱数据

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
#include "utils.h" // For printContainer(): see 7.11

using namespace std;

int main() {

   vector<int> v;
   back_insert_iterator<std::vector<int> > p =
      back_inserter(v);

   for (int i = 0; i < 10; ++i)
     *p = i;

   printContainer(v, true);

   random_shuffle(v.begin(), v.end());

   printContainer(v, true);
}
```

输出如下所示：

```
-----
0 1 2 3 4 5 6 7 8 9
-----
8 1 9 2 0 5 7 3 4 6
```

讨论

random_shuffle函数使用起来很直接。给定一个范围，它将在这个范围内随机打乱这些元素。它有两个版本，其原型如下：

```
void random_shuffle(RndIter first, RndIter last);
void random_shuffle(RndIter first, RndIter last, RandFunc& rand);
```

在这第一个版本中，这个随机是指使用一个特殊实现的随机数产生函数，这个对于你的大部分需求是满足的。如果不满足，你可以写一个非均匀的产生函数，如高斯算法，你可以写出你自己的随机数产生函数来使用第二个版本。

www.TopSage.com

大家网 TopSage.com

你的随机数产生器必须是一个算符，它带一个简单参数并返回一个简单值，这两个值都能转换成iterator_traits<RndIter>::difference_type类型。在大部分情况下，一个整型数就可以。例如，下面就是我的一个随机数产生器：

```
struct RanNumGenFtor {
   size_t operator()(size_t n) const {
 return(rand() % n);
   }
} rnd;

random_shuffle(v.begin(), v.end(), rnd);
```

使用random_shuffle函数限制在那些提供了随机访问迭代器的序列中（字符串、矢量和双端队列），数组或者你的实现随机访问迭代器的容器。你不能随机搅乱一个集合容器，因为它的内容是以某种顺序存储的。事实上，你不能在一个集合容器上使用任何算法（常常作为一个变异的算法）来修改它的范围。

7.4 比较元素

问题

有两个序列，如何比较它们是否是相等或是基于某种元素的顺序，如何给这两个序列排序？

解决方案

基于你想使用的比较的类型，你可以使用定义在<algorithm>文件中的equal、lexicographical_compare或者 mismatch标准算法。示例7-4给出了几个不同的比较例子。

示例 7-4 不同种类的比较

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include "utils.h"

using namespace std;
using namespace utils;

int main() {

   vector<string> vec1, vec2;

   vec1.push_back("Charles");
   vec1.push_back("in");
   vec1.push_back("Charge");

   vec2.push_back("Charles");
   vec2.push_back("in");
   vec2.push_back("charge");  // Note the small "c"

   if (equal(vec1.begin(), vec1.end(), vec2.begin())) {
      cout << "The two ranges are equal!" << endl;
```

www.TopSage.com

```
    } else {
       cout << "The two ranges are NOT equal!" << endl;
    }

    string s1 = "abcde";
    string s2 = "abcdf";
    string s3 = "abc";

    cout << boolalpha  // Show bools as "true" or "false"
         << lexicographical_compare(s1.begin(), s1.end(),
                                    s1.begin(), s1.end()) << endl;
    cout << lexicographical_compare(s1.begin(), s1.end(),
                                    s2.begin(), s2.end()) << endl;
    cout << lexicographical_compare(s2.begin(), s2.end(),
                                    s1.begin(), s1.end()) << endl;
    cout << lexicographical_compare(s1.begin(), s1.end(),
                                    s3.begin(), s3.end()) << endl;
    cout << lexicographical_compare(s3.begin(), s3.end(),
                                    s1.begin(), s1.end()) << endl;

    pair<string::iterator, string::iterator> iters =
       mismatch(s1.begin(), s1.end(), s2.begin());

    cout << "first mismatch  = " << *(iters.first) << endl;
    cout << "second mismatch = " << *(iters.second) << endl;
}
```

示例7-4的输出如下所示：

```
The two sequences are NOT equal!
false
true
false
false
true
first mismatch  = e
second mismatch = f
```

讨论

使用equal算法比较两个序列是否相等。它带三个或四个参数，这依赖于你使用的版本。下面是它们的声明：

```
bool equal(In1 first1, In1 last1, In2 first2);
bool equal(In1 first1, In1 last1, In2 first2, BinPred pred);
```

equal算法使用操作符==来用从first1和last1之间的每个元素与从first2开始的每个元素进行比较。如果你提供了pred参数的话，equal算法就用这个参数测试它们是否相等。在调用这个equal算法之前必须保证这两个序列具有相同的长度；或者第二个序列的长度不比第一个序列的长度短，否则的话，这个算法的行为就是不可靠的。

你如果想知道两个序列不同的具体信息的话，你就可以使用lexicographical_compare或者mismatch算法，lexicographical_compare算法比较两个序列并且如果第一个序列按照字典顺序小于第二个的话，就返回真。这就意味在这两个序列中的每一对比较使用的是<操作符。这个lexicographical_compare比较算法的声明如下：

```
bool lexicographical_compare(In1 first1, In1 last1,
                             In2 first2, In2 last2);
```

www.TopSage.com

```
bool lexicographical_compare(In1 first1, In1 last1,
                             In2 first2, In2 last2,
                             Compare comp);
```

只要操作符<返回真，或者第一个序列在第二个序列之前到达末尾，这个算法就返回真。否则就返回假。如示例7-4中的两个字符序列的比较：

```
string s1 = "abcde";
string s2 = "abcdf";
string s3 = "abc";

lexicographical_compare(s1.begin(), s1.end(),     // abcde < abcde
                        s1.begin(), s1.end());    // = false
lexicographical_compare(s1.begin(), s1.end(),     // abcde < abcdf
                        s2.begin(), s2.end());    // = true
lexicographical_compare(s2.begin(), s2.end(),     // abcdf < abcde
                        s1.begin(), s1.end());    // = false
lexicographical_compare(s1.begin(), s1.end(),     // abcde < abc
                        s3.begin(), s3.end());    // = false
lexicographical_compare(s3.begin(), s3.end(),     // abc < abcde
                        s1.begin(), s1.end());    // = true
```

lexicographical_compare算法的复杂度是线性的，并且它只做这两个序列中常度更小的那个序列的长度个数的比较，或者它的比较在其中的一个序列中的任意一个元素第一次小于另一个序列中对应元素时就结束。这个比较的实现都是使用操作符<，因此如果iter1和iter2分别是这两序列迭代器的话，那么只要*iter1小于*iter2或者*iter2小于*iter1中任意一个成立，比较就会停止。

mismatch算法将会告诉你这两个序列在哪不同。它的声明与equal和lexicographical_compare算法都有点不同，因为它返回一个迭代器对<>而不是一个布尔值。声明如下所示：

```
pair<In1, In2> mismatch(In1 first1, In1 last1, In2 first2);
pair<In1, In2> mismatch(In1 first1, In1 last1, In2 first2, BinPred);
```

这两个迭代器都返回一个指向这两个序列中元素的指针。如示例7-4：

```
string s1 = "abcde";
string s2 = "abcdf";
pair<string::iterator, string::iterator> iters =
   mismatch(s1.begin(), s1.end(), s2.begin());

cout << "first mismatch  = " << *(iters.first) << '\n'; // 'e'
cout << "second mismatch = " << *(iters.second) << '\n';// 'f'
```

你必须保证第二个序列的长度不比第一个小。如果第二个序列的长度比第一个小的话，mismatch就没办法知道并且它会继续用第二个序列中末端之后的元素进行比较，而这种情况下，这个算法的行为是不确定的。另外，如果在这个比较中不存在比较不成功的比较的话，那么这第一个迭代器将指向last1，而它可能是不合法的（例如，你如果用end()函数作为last1参数传进去的话）。

你或许已经从每一个这种算法的声明注意到了这两个序列中每一个的迭代器的类型都是不同的。这也就意味着这两个序列可以是不同类型的容器，只要这种类型的迭代器所指

www.TopSage.com

向的元素都定义了操作符<就可以。例如，你可以用一个字符串与一个vector<char>进行比较：

```
string s = "Coke";
vector<char> v;

v.push_back('c');
v.push_back('o');
v.push_back('k');
v.push_back('e');

std::cout << std::lexicographical_compare(s.begin(), s.end(),
             v.begin(), v.end()) << '\n';
```

这将比较两个序列中的每一个元素而不会考虑它们的容器是什么类型。

C++的标准类库给序列的比较提供几种不同的方法。如果没有一种适合你的需求，请参考一下它们的源代码；它将会给你如何实现一个你自己的高效的通用的算法提供一个很好的例子。

请参考7.1节。

7.5 合并数据

问题

有两个已排序的序列，如何合并它们？

解决方案

使用merge或者inplace_merge函数模板。merge函数模板合并两个序列并把合并后的结果存到第三个序列中去，而inplace_merge则是合并两个连续的序列。示例7-5给出了一个简单的例子。

示例7-5 合并两个序列

```
#include <iostream>
#include <string>
#include <list>
#include <vector>
#include <algorithm>
#include <iterator>
#include "utils.h" // For printContainer(): see 7.10

using namespace std;

int main() {

   vector<string> v1, v2, v3;

   v1.push_back("a");
   v1.push_back("c");
   v1.push_back("e");
```

www.TopSage.com

```
    v2.push_back("b");
    v2.push_back("d");
    v2.push_back("f");

    v3.reserve(v1.size() + v2.size() + 1);

    // Use a back_inserter from iterator to avoid having to put
    // a bunch of default objects in the container.  But this doesn't
    // mean you don't have to use reserve!
    merge(v1.begin(), v1.end(),
          v2.begin(), v2.end(),
          back_inserter<vector<string> >(v3));

    printContainer(v3);

    // Now make a mess
    random_shuffle(v3.begin(), v3.end());
    sort(v3.begin(), v3.begin() + v3.size() / 2);
    sort(v3.begin() + v3.size() / 2, v3.end());

    printContainer(v3);

    inplace_merge(v3.begin(), v3.begin() + 3, v3.end());

    printContainer(v3);

    // If you are using two lists, though, use list::merge instead.
    // As a general rule, blah blah...
    list<string> lstStr1, lstStr2;

    lstStr1.push_back("Frank");
    lstStr1.push_back("Rizzo");
    lstStr1.push_back("Bill");
    lstStr1.push_back("Cheetoh");

    lstStr2.push_back("Allie");
    lstStr2.push_back("McBeal");
    lstStr2.push_back("Slick");
    lstStr2.push_back("Willie");

    lstStr1.sort();  // Sort these or merge makes garbage!
    lstStr2.sort();

    lstStr1.merge(lstStr2); // Note that this only works with other
                            // lists of the same type

    printContainer(lstStr1);
}
```

示例7-5输出如下：

```
-----
a
b
c
d
e
f
-----
b
d
e
a
c
f
-----
```

www.TopSage.com

大家网 TopSage.com

```
a
b
c
d
e
f
-----
Allie
Bill
Cheetoh
Frank
McBeal
Rizzo
Slick
Willie
```

讨论

merge函数合并两个已排序的序列并把合并的结果存到第三个序列中去，并且用户可以提供一个比较算符来决定一个元素比另一个元素小，这个比较算符默认使用<操作符。算法的复杂度是线性的：merge 函数执行的比较次数是两个序列长度之和减1。每个序列中的元素的类型必须与操作符<兼容（或者与你提供的算符兼容），并且通过复制构造函数或赋值操作，它们一定能转换成输出序列的元素类型；或者必须有一个定义好的转换操作符，即在输出序列中的元素类型必须给这两个类型都定义赋值操作和复制构造函数。

merge 函数的声明如下：

```
void merge(In1 first1, In1 last1, In2 first2, In2 last2, Out result)
void merge(In1 first1, In1 last1, In2 first2, In2 last2, Out result,
           BinPred comp)
```

merge 函数的使用是比较简单的。这两个序列必须是排过序的（否则输出的结果就没有任何意义），并且 merge 函数不会修改这两个序列。用来存储结果的输出迭代器必须有足够大的空间来存储这两个序列中的元素。你可以直接使用reserve函数来申请足够的存储空间，或者使用 back_inserter 函数，就像我在示例 7-5 中那样：

```
merge(v1.begin(), v1.end(),
      v2.begin(), v2.end(),
      back_inserter(v3));
```

一个 back_inserter 是一个定义在 <iterator> 中的类，它提供了一个方便的方法来创建一个输出迭代器，每一次你给它赋一个值时，它都调用 push_back 函数来把这个值加到序列中去。这样，你就不用显式地指出输出序列的大小。以下就是调用 back_inserter 给一个名为 v3 的 vector<string> 创建的输出序列：

```
back_inserter(v3);
```

你不用显式地指出模板参数的类型，因为back_inserter是一个函数模板而不是一个类模板，因此这个调用参数的类型是可以省略的。和上面等价的一个调用形式如下：

```
back_inserter<vector<string> >(v3);
```

www.TopSage.com

然而，有时你还是应该显式地指明输出序列的大小，特别是输出序列是一个矢量时。如果你简单地调用push_back而不指明序列的大小，这个矢量可能需要自动重新调整大小，而这个操作是一个昂贵的操作。更多的关于矢量的信息请参考6.2节。

在这两个序列中如果有两个元素相同，那么来自第一个序列中的元素将优先于来自第二序列中的元素。因此，如果你交换这两个输入序列来调用merge函数两次的话，那么这两次的输出序列可能不同（虽然你希望它们相同，但是结果是不同的）。

合并链表是你既可以用成员函数又可以使用功能相同的标准算法的一个很好的例子。在这种情况下，你应该优先使用成员函数而不是使用标准算法，但也不总是这样，下面就是一个例子：

考虑来自示例7-5中的字符串序列：

```
lstStr1.sort();  // Sort these or merge makes garbage!
lstStr2.sort();
lstStr1.merge(lstStr2); // This is list::merge
```

在这种情况下不调用std::merge函数有两个原因。首先，这两个链表必须有相同的元素类型。这是因为list::merge的声明如下：

```
void merge(list<T, Alloc>& lst)
template <typename Compare>
void merge(list<T, Alloc>& lst, Compare comp)
```

T与这个list类模板本身具有同样的类型。因此，你不能这样做，例如，合并一个以null结尾的字符数组到这个字符串链表中。

另外一个就是list::merge函数将会删除输入序列，而std::merge则不会修改这两个输入序列。更为重要的是，list::merge函数具有更好的性能，因为在大部分情况下，链表中的元素都是重新连接一下而不是复制；但这个重新连接是不能肯定的，因此你需要查看一下源代码或做个试验来验证一下。

你可以使用inplace_merge函数来合并两个连续的序列。inplace_merge函数和merge函数是不同的，因为它合并它的两个序列参数。换句话说，如果你有多个连续的序列（例如，它们是同一个序列的两部分），并且它们排过序，并且你又想把整个序列排序的话，你就可以使用inplace_merge函数而不是使用sort算法。它的优点是如果有足够的内存空间的话，它的复杂度是线性的；如果没有的话，那么它的复杂度是$n\log n$数量级的，而这是排序算法的一个平均复杂度。

inplace_merge的声明和merge函数有一点不同，如下所示：

```
void inplace_merge(Bid first, Bid mid, Bid last)
void inplace_merge(Bid first, Bid mid, Bid last, BinPred comp)
```

inplace_merge函数需要双向迭代器，因此你不能把它与merge函数交换使用，但是

www.TopSage.com

部分情况下是能工作的。同merge函数一样，它也使用<操作符作为默认的操作符来决定元素的顺序，并且你也可以提供一个比较函数comp给它。

7.6 排列元素

问题

如何给一定范围内的元素排序?

解决方案

有很多可以用来给一个范围内元素排序的算法。你可以使用传统的排序(升序或降序)算法sort函数，它定义在<algorithm>文件中，或者你可以使用其他的一些排序函数，如partial_sort。请看示例7-6中的实现：

示例7-6 排序

```
#include <iostream>
#include <istream>
#include <string>
#include <list>
#include <vector>
#include <algorithm>
#include <iterator>
#include "utils.h" // For printContainer(): see 7.10

using namespace std;

int main() {

   cout << "Enter a series of strings: ";
   istream_iterator<string> start(cin);
   istream_iterator<string> end; // This creates a "marker"
   vector<string> v(start, end);

   // The sort standard algorithm will sort elements in a range.  It
   // requires a random-access iterator, so it works for a vector.
   sort(v.begin(), v.end());
   printContainer(v);

   random_shuffle(v.begin(), v.end()); // See 7.2

   string* arr = new string[v.size()];

   // Copy the elements into the array
   copy(v.begin(), v.end(), &arr[0]);

   // Sort works on any kind of range, so long as its arguments
   // behave like random-access iterators.
   sort(&arr[0], &arr[v.size()]);
   printRange(&arr[0], &arr[v.size()]);

   // Create a list with the same elements
   list<string> lst(v.begin(), v.end());

   lst.sort(); // The standalone version of sort won't work; you have
               // to use list::sort.  Note, consequently, that you
               // can't sort only parts of a list.
```

www.TopSage.com

```
    printContainer(lst);
}
```

示例7-6中的输出如下：

```
Enter a series of strings: a z b y c x d w
^Z
-----
a b c d w x y z
-----
w b y c a x z d
-----
a b c d w x y z
-----
a b c d w x y z
```

讨论

排序是一个通常的需求，并且这有两种方法可用来给一个序列排序。你可以使用一个集合容器从而就以排序的方式保存元素，但是当插入元素时，你就得付出重新排序的时间上的代价，或者你可以按需排序，对这种情况，你可以有多种选择。

标准的排序算法的行为和你希望的相同：它在一定的范围内使用操作符 < 来以升序或者降序排列元素。它的声明如下：

```
void sort(Rnd first, Rnd last);
void sort(Rnd first, Rnd last, BinPred comp);
```

就像大部分算法一样，当默认的 < 操作符不能满足你的需求时，你也可以给它提供一个排序用的比较操作符。平均来说，它的复杂度是 $n \log n$ 数量级。但最坏的情况下是 n 的平方数量级。

如果你需要使得相等的元素排序以后还保持相对的位置，你就可以使用 stable_sort 函数。它和前面的 sort 函数具有相同的函数签名，但它能保证相等的元素排序完后的相对位置不发生改变。它的复杂度有些不同，因为它在最坏的情况下是 $n \log n$ 数量级，只要有足够的内存空间。如果没有足够的内存空间的话，那么它的复杂度最多也就是 $n (\log n)^2$。

sort 函数并不是在任意的容器中都可以工作。它需要一个随机访问的迭代器，如果你使用的容器不提供这种类型的迭代器的话，它将不能正常工作。标准的序列容器如双端队列、矢和字符串（string/wstring）（它们都不是容器，但它们满足序列容器的大部分要求），它们都提供了这种随机访问迭代器。但链表 list 没有提供。因此，如果你需要给链表排序时，你需要使用 list::sort 算法。例如，在示例7-6中使用了不带参数的 list::sort 函数：

```
lst.sort();
```

它和 std::sort 是不同的，因为你不能给链表的一部分排序。因此如果你需要给一个序列中的某个部分排序的话，你最好使用序列而不是链表。

www.TopSage.com

排序的概念是很好理解的，但在标准类库的实现中也有很多变种。一一描述如下：

partial_sort

这个函数带三个随机访问迭代器：first、middle和last，并且可以选择是否带一个比较算符。它有两个后置条件：在（first，middle）范围中的元素比在（middle，last）范围中的元素都小，并且在（first，middle）范围中的元素中的元素根据操作符<或你提供的比较算符来排序。换句话说，它一直排序直到第一批*n*个元素都排完序为止。

partial_sort_copy

这个函数与partial_sort具有相同的功能，但它把结果放到一个输出范围中去。它把排完序的第一批*n*个元素从原范围中以排序后的顺序复制到目标范围中。如果目标范围（*n*）比原范围（*m*）小的话，那么只有*n*个元素被复制到目标范围中。

nth_element

这个函数带有三个随机访问迭代器作为参数：first、nth和last，并且带一个可选的比较算符。它把nth指向的元素放到这个范围排完序后它应该在的那个位置处。因此所有在（first，nth）范围中的元素都比nth这个位置（所有这些在（nth，last）范围中的元素都没排序，但它们都比nth指向的元素前面的元素都大）指向的元素都小。如果你只需要对一个范围内的某几个元素排序的话，你就可以使用这个函数，这样你就可以不用对整个范围内的元素排序。

根据你提供的标准（funtor算符），你也可在这个范围中分割这些元素，这将在7.7节中讨论。

请参考7.7节。

7.7 分割元素

问题

如何以某种定义好的方式对一定范围内的元素进行分割？例如，你可能需要把小于某个特殊值的所有元素移到序列的前面去。

解决方案

用一个断言算符与partition标准算法一起使用来移动这些元素。请看示例7-7。

示例7-7 分割一个范围

```
#include <iostream>
#include <istream>
#include <string>
#include <vector>
#include <algorithm>
```

www.TopSage.com

大家网 TopSage.com

```
#include <functional>
#include <iterator>
#include "utils.h" // For printContainer(): see Recipe 7.10

using namespace std;

int main( ) {

   cout << "Enter a series of strings: ";
   istream_iterator<string> start(cin);
   istream_iterator<string> end; // This creates a "marker"
   vector<string> v(start, end);

   // Rearrange the elements in v so that those that are less
   // than "foo" occur before the rest.
   vector<string>::iterator p =
      partition(v.begin(), v.end(),
                bind2nd(less<string>(), "foo"));
   printContainer(v);

   cout << "*p = " << *p << endl;
}
```

示例 7-7 的输出如下：

```
Enter a series of strings: a d f j k l
^Z
-----
a d f j k l
*p = j
```

使用partition分割函数后，这个迭代器p指向那些使得less(*p, "foo")不为真的元素中的第一个元素。

讨论

partition函数带三个参数：范围的开始、范围的结束和一个断言，并且它把那些使得这个断言为真的所有元素都移到这个范围的前面。然后它返回一个迭代器，并且这个迭代器指向那些使得这个断言不为真的所有元素的第一个元素，或者如果这个范围中的所有元素都满足这个断言的话，这个迭代器就指向这个范围的结束。它的声明如下：

```
Bi partition(Bi first, Bi last, Pred pred);
```

pred就是这个算符，它带有一个参数并返回真或假。并且这个分割函数没有默认的断言；也就是你必须给你正在分割的范围提供一个断言才有意义。你可以实现你自己的断言或者使用标准类库中提供的断言。例如，示例 7-7 中，你可以看到我使用了 less 和 bind2nd 函数来创建一个算符：

```
vector<string>::iterator p =
   partition(v.begin(), v.end(),
             bind2nd(less<string>(), "foo"));
```

它所做的就是把比"foo"小的所有元素移到那些不比 “foo”小的元素前面，这里bind2nd不是必须的，但它是一个自动创建一个算符的方便的方法，这个算符带有一个参数并且返回这个序列中每一个元素 i 在 less<string>(*i, "foo")中的结果。如果你希望相等的元素还保持它们相对的顺序的话，你可以使用 stable_partition 函数。

www.TopSage.com

注意： partition和其他的能改变范围中元素顺序的算法在这些标准的集合容器（如set、multiset、map和multimap）中是不能正常工作的。这是因为这些集合容器都以一个定义好的顺序来保存它们的元素并且只有这些容器本身才允许移动或删除它们的元素。你可以在任何你最少能获取到双向迭代器的序列上使用partition算法，这些序列包括标准序列容器双端队列、矢量和链表。

请参考7.9节。

7.8 在序列上进行集合运算

问题

如何在你的序列上使用一些集合运算，如union、difference或intersection?

解决方案

使用标准类库函数就能达到这个目标：set_union、set_difference和set_intersection。它们每一个都执行它们各自的集合操作并把结果放到一个输出范围中。请看示例7-8。

示例7-8 使用集合操作

```
#include <iostream>
#include <algorithm>
#include <string>
#include <set>
#include <iterator>
#include "utils.h" // For printContainer(): see 7.10

using namespace std;

int main() {

   cout << "Enter some strings: ";
   istream_iterator<string> start(cin);
   istream_iterator<string> end;
   set<string> s1(start, end);

   cin.clear();

   cout << "Enter some more strings: ";
   set<string> s2(++start, end);

   set<string> setUnion;
   set<string> setInter;
   set<string> setDiff;

   set_union(s1.begin(), s1.end(),
             s2.begin(), s2.end(),
             inserter(setUnion, setUnion.begin()));

   set_difference(s1.begin(), s1.end(),
                  s2.begin(), s2.end(),
                  inserter(setDiff, setDiff.begin()));
```

www.TopSage.com

```
    set_intersection(s1.begin(), s1.end(),
                     s2.begin(), s2.end(),
                     inserter(setInter, setInter.begin()));

    cout << "Union:\n";
    printContainer(setUnion);
    cout << "Difference:\n";
    printContainer(setDiff);
    cout << "Intersection:\n";
    printContainer(setInter);
}
```

这个程序的输出如下（printContainer打印出一个容器中的内容）：

```
Enter some strings: a b c d
^Z
Enter some more strings: d e f g
^Z
Union: a b c d e f g
Difference: a b c
Intersection: d
```

讨论

标准类库中的集合操作基本上都相同。每个操作都带有两个范围，分别在它们上面执行各自的操作，并且把结果保存到一个输出迭代器中。在这个输出序列中你不得不保证它有足够的空间，或者使用inserter或者使用back_inserter（如何使用back_inserter，请参考7.5节）。

set_union的声明如下：

```
Out set_union(In first1, In last1, In first2, In last2, Out result)
```

set_difference、set_intersection和set_symmetric_difference的声明与这个声明相似。

你可以像我在示例7-8中那样使用这些函数。例如，为了找到两个集合的交集，你可以和下面一样调用set_intersection函数：

```
set_intersection(s1.begin(), s1.end(),
                 s2.begin(), s2.end(),
                 inserter(setInter, setInter.begin()));
```

set_intersection函数的最后一个参数需要进一步的解释。inserter是一个定义在<iterator>中的函数模板，它带有一个容器和迭代器作为参数并且返回一个输出迭代器，当有值赋给它时，它就调用insert函数在它的第一个参数上插入这个值。如果你在一个序列容器上使用这个函数的话，它就在你作为它的最后一个参数传给它的迭代器前面插入这个值。如果你在一个集合容器上使用它，就像我在前面的代码中那样，这个迭代器将被忽略并且将依据这个容器的排序标准来插入这些元素。

就我这个目的而言，集合是一个方便的例子，但是你可以在任意的序列上调用集合操作，而不只是在集合容器上。例如，或许有链表也需要使用集合操作：

www.TopSage.com

```
list<string> lst1, lst2, lst3;

// Fill them with data

lst1.sort();// Elements must be sorted
lst2.sort();

set_symmetric_difference(lst1.begin(), lst1.end(),
                         lst2.begin(), lst2.end(),
                          back_inserter(lst3));
```

然而，由于链表并不是按一定的顺序来排序，所以你不得不首先自己给它们排序，否则这些集合操作的结果就是不可靠的。请注意在这个例子中，我使用的是back_inserter而不是inserter。back_inserter的工作方式和inserter差不多，但这个函数调用push_back来给你的容器追加元素。你可以不这样做；例如，你可以使用resize来重新调整容器的大小使得它一直都具有足够的空间：

```
lst3.resize(lst1.size() + lst2.size());

set_symmetric_difference(lst1.begin(), lst1.end(),
                         lst2.begin(), lst2.end(),
                         lst3.begin());
```

如果这个输出序列足够大的话，你可以使用begin来传一个指向这个序列中的第一个元素的迭代器。

也许你不太了解set_symmetric_difference这个函数的功能，这里我就解释一下。如果a和b都是一个集合的话，它就是a - b 或 b -a。换句话说，它的值是一个集合中的所有不在另一个集合中的元素。

关于集合操作你还需要知道的一点就是，由于序列中的值不是唯一的，因此你就可能在这个集合中具有重复的值。严格地说，当然，数学意义上的集合是不允许重复的值的，所以这种集合就不是严格意义上的集合。在示例7-8中，如果我使用链表而不是集合的话，它的输出如下（当你运行示例7-8时，你可以输入重复的值，但是它们不会被加到集合中去，因为当你插入一个已经在集合中存在的值时，set::insert函数会失败）：

```
Enter some strings: a a a b c c
^Z
Enter some more strings: a a b b c
^Z
Union: a a a b b c c
Difference: a c
Intersection: a a b c
```

这里所做的就是这些集合操作枚举这两个序列并且比较相应的值，从而决定什么值应该加到输出序列中去。

最后，默认形式的集合操作（使用操作符<来比较元素）与你的集合包含的指针的工作方式不同。为了搞清楚这个区别，在7.4节中，我写了一个函数来比较指针对象。

请参考7.4节。

www.TopSage.com

7.9 序列元素转换

问题

你有一系列的元素，如何在每一个元素上做某种操作，移动它或者把它复制到另一个序列中去？

解决方案

你可以使用transform或for_each标准算法。这两个算法都很简单，它们几乎能让你给你序列中的每个元素执行任意的操作。示例7-9是一个简单例子。

示例7-9 数据转换

```
#include <iostream>
#include <istream>
#include <string>
#include <list>
#include <algorithm>
#include <iterator>
#include <cctype>
#include "utils.h" // For printContainer(): see 7.10

using namespace std;

// Convert a string to upper case
string strToUpper(const string& s) {
   string tmp;
   for (string::const_iterator p = s.begin(); p != s.end(); ++p)
      tmp += toupper(*p);
   return(tmp);
}

string strAppend(const string& s1, const string& s2) {
   return(s1 + s2);
}

int main() {

   cout << "Enter a series of strings: ";
   istream_iterator<string> start(cin);
   istream_iterator<string> end;
   list<string> lst(start, end), out;

   // Use transform with an unary function...
   transform(lst.begin(), lst.end(),
             back_inserter(out),
             strToUpper);

   printContainer(out);

   cin.clear();

   cout << "Enter another series of strings: ";
   list<string> lst2(++start, end);
   out.clear();

   // ...or a binary function and another input sequence.
   transform(lst.begin(), lst.end(), lst2.begin(),
             back_inserter(out),
             strAppend);
```

www.TopSage.com

大家网 TopSage.com

```
    printContainer(out);
}
```

讨论

数据转换的一个明显的函数就是transform，它有两个形式。第一种形式就是它带一个序列、一个输出迭代器和一个一元算符。它把这个一元算符应用在这个序列上的每一个元素上，并且把返回的值赋给这个迭代器指向的下一个元素。这个输出迭代器可以是另外一个序列或者是原有的这个序列的开始。在这个方面，transform函数既处理复制类型又处理位置变换。

transform的声明如下：

```
Out transform(In first, In last, Out result, UnFunc f);
Out transform(In first1, In last1, In first2, In last2,
              Out result, BinFunc f);
```

这两个版本都返回一个指向结果序列末端的后面一个。

这两个版本使用起来都是非常直接的。为了把一个字符串从一个序列以大写字母复制到另一个序列中，可以与示例7-9一样来处理：

```
std::transform(lst.begin(), lst.end(),
               std::back_inserter(out), strToUpper);
```

如果你想修改这个原有的序列，你就可以仅用这个序列开始传给这个结果迭代器：

```
std::transform(lst.begin(), lst.end(),
               lst.begin(), strToUpper);
```

使用两个序列和一个二元操作符也是一样，你可以用这两个输入序列中的任意一个作为这个输出序列。

如果你想在原有的位置上转换元素，你就应避免把每个元素赋给一些函数的返回值。或者你想使用这个算符去改变它的原对象，你可以使用for_each：

```
void strToUpperInPlace(string& s) {
   for (string::iterator p = s.begin(); p != s.end(); ++p)
      *p = std::toupper(*p);
}
// ...
std::for_each(lst.begin(), lst.end(), strToUpperInPlace);
```

如果你想做的是改变序列本身而不是改变每个元素的话，参考7.6节，在这一节中我描述了很多改变序列中元素位置的标准算法。

请参考7.1节和7.6节。

www.TopSage.com

7.10 编写自己的算法

问题

你需要在一个范围中执行一个算法，但是没有标准的算法满足你的需求，如何编写你自己的算法？

解决方案

使用函数模板来实现你的算法并推荐使用模板参数的名字来列举你的需求。请看示例7-10中copy标准算法的一个变体。

示例7-10 实现你自己的算法

```
#include <iostream>
#include <istream>
#include <iterator>
#include <string>
#include <functional>
#include <vector>
#include <list>
#include "utils.h" // For printContainer(): see 7.11

using namespace std;

template<typename In, typename Out, typename UnPred>
Out copyIf(In first, In last, Out result, UnPred pred) {
   for ( ;first != last; ++first)
      if (pred(*first))
         *result++ = *first;
   return(result);
}

int main() {

   cout << "Enter a series of strings: ";
   istream_iterator<string> start(cin);
   istream_iterator<string> end; // This creates a "marker"
   vector<string> v(start, end);

   list<string> lst;

   copyIf(v.begin(), v.end(), back_inserter<list<string> >(lst),
      bind2nd(less<string>(), "cookie"));

   printContainer(lst);
}
```

示例7-10的输出如下：

```
Enter a series of strings: apple banana danish eclaire
^Z
-----
apple banana
```

你可能注意到了只有一部分复制到了目标范围中。

www.TopSage.com

讨论

标准类库包含一个copy函数模板，它把元素从一个范围复制到另一个范围，但是标准类库中的copy函数模板没有带一个断言作为参数并依据这个条件来复制每个元素的版本（例如，一个copy_if算法），因此这就是示例7-10中的实现。它的功能也很简单：给出一个原范围和一个目标范围的开始处，然后把这些使得这个一元断言为真的元素复制到目标范围中。

虽然这个算法简单，但是还是有很多可以讨论的东西在里头。从它的声明，你就能发现它有三个模板参数，如下所示：

```
template<typename In, typename Out, typename UnPred>
Out copyIf(In first, In last, Out result, UnPred pred) {
```

第一个模板参数In是这个输入迭代器的类型。由于这是一个输入范围，copyIf函数需要做的全部工作就是从这个迭代器中索引出相应的值并且增加这个迭代器使得它指向下一个元素。这说明了这个输入迭代器的类型（迭代器的类型在7.1节中已经说明了），因此根据这个迭代器的类型，我推荐你给这个模板参数命名为In。这没有标准的惯例（In和Out或许是我的惯例，在7.1节中我做了说明），但通过这个类似的命名惯例能够很好地传达你的意图：InIter、Input_T或者InputIterator。第二个模板参数Out是这个指向所有的元素将要被复制到的目标范围的迭代器的类型。copyIf需要能够向这个迭代器写入它索引到的值并增加这个迭代器，它就是这个输出迭代器。通过模板参数的名字就传达了你的迭代器需求，就可以使用一个自说明型算法的调用惯例。但为什么需要使用两个不同的迭代器类型呢？

有很多原因使得我在copyIf函数中使用两个不同类型的迭代器。首先，在每个范围上操作有些不同，并且由于在输入范围中我从来没有需要使用向后的操作，或者是给它赋值，所有我需要的就是一个输入迭代器。同样，我将肯定不需要从输出范围中读取数据，所以这里我所需要的就是一个输出迭代器。对每个迭代器来说，这里都不需要它应用到对方身上，所以使用双向迭代器类型就没有意义。第二，使用这两个不同的迭代器类型能让用户从一个这种范围中读取数据并写入到另一个范围中。示例7-10中，我从一个矢量中读取并写到另一个链表中：

```
vector<string> v(start, end);
list<string> lst;

copyIf(v.begin(), v.end(), back_inserter<list<string> >(lst),
   bind2nd(less<string>(), "cookie"));
```

如果你试着只使用一种类型的迭代器，它将不能被编译成功。

示例7-10中，我传递了一个back_inserter来代替输出范围的开始处，也就是说，传递了一个从lst.begin返回的迭代器。我这样做是因为lst没有元素，并且在这个算法中（就像在copy标准算法），这个目标范围不得不具有足够大的空间来容纳下所有的

www.TopSage.com

要复制到它的元素。否则，copyIf函数中增加这个输出迭代器将就会导致一个不可靠的行为。一个back_inserter在你增加这个迭代器时将返回一个在这个容器上调用push_back函数的输出迭代器。每次这个输出迭代器结果增加就使得1st的容量也增加了。在7.5节中我详细描述了back_inserter类模板。

当你在一个范围（例如，标准容器）上写你自己的算法时，你应该多考虑用迭代器作为参数而不是用容器作为参数。你也许有兴趣试一下，例如，把copyIf声明为带两个容器参数而不是使用原范围和目的范围输出迭代器，但这种解决方案的通用性较差。如果你使用容器参数的话，你就不能为这个容器内的所有元素的子集使用这个算法。更为重要的是，在这个copyIf函数体中，你将需要依赖这个容器的begin和end成员函数来得到你正在找的范围，并且它的返回类型也将依赖这个作为输出范围的容器类型。这就意味着copyIf不能在非标准范围上工作，例如，一个内置的数组或者你定制的容器。这些原因加上一些别的原因就是为什么标准算法都操作在范围上的原因。

最后，如果你确实需要写出你自己的算法，你应该用你的需求仔细检查一下标准算法。第一眼也许你会觉得它们非常简单，但是就是因为它们的通用性才使得它们简单，并且十次有九次通过扩展它们就可以满足你的需求。重用这些标准算法是你最应该做的，因为它们在移植性和效率上是有保证的。

请参考7.5节。

7.11 把元素输出到数据流

问题

你有一个范围的元素，为了调试的目的，如何把它们输出到一个流中去？

解决方案

写一个带一个范围或一个容器参数的函数模板，枚举它的每一个元素，并使用copy算法和ostream_iterator算法把每一个元素都写到一个流中去。如果你在输出的格式上想有更多的控制，你可以写一个你自己的简单算法来枚举一个范围并把每一个元素都打印到流中去（见示例7-11）。

示例7-11 在一个流中打印一个范围内的元素

```
#include <iostream>
#include <string>
#include <algorithm>
#include <iterator>
#include <vector>

using namespace std;

int main() {
```

www.TopSage.com

```
    // An input iterator is the opposite of an output iterator: it
    // reads elements from a stream as if it were a container.
    cout << "Enter a series of strings: ";
    istream_iterator<string> start(cin);
    istream_iterator<string> end;
    vector<string> v(start, end);

    // Treat the output stream as a container by using an
    // output_iterator.  It constructs an output iterator where writing
    // to each element is equivalent to writing it to the stream.
    copy(v.begin(), v.end(), ostream_iterator<string>(cout, ", "));
}
```

示例7-11的输出如下：

```
Enter a series of strings: z x y a b c
^Z
z, x, y, a, b, c,
```

讨论

一个流迭代器是一个基于流而不是基于某个容器中的某个范围内的元素的迭代器，并且流迭代器让你能把一个流的输入当成一个输入迭代器（从索引到的值读并增加这个迭代器）或者当成一个输出迭代器（与输入迭代器差不多，但是它是向这个索引到的值写而不是从它读）。这就使得从一个流中读值（特别是字符串）变得很简洁，如本章的其他的很多例子中那样，并且向一个流中写也很简洁，就像在示例7-11中那样。我知道这一节的主题是关于把一个范围内元素打印到一个流中去的，但是我还是要花一些时间来解释一下输入流迭代器，因为在本章的很多例子中我都使用了它。

示例7-11中istream_iterator有三个关键的部分。第一部分是创建这个istream_iterator并使它指向一个流输入的开始处。我的做法如下所示：

```
istream_iterator<string> start(cin);
```

这将创建一个名为start的指向一个输入序列第一个元素的迭代器，就像vec.begin（vec是一个矢量）一样返回指向这个矢量的第一个元素的迭代器。这个模板参数告诉istream_iterator这个序列中的元素都是字符串。这个构造函数参数cin是用来读取的输入流。这是一个抽象，因为在这个点上还没有第一个元素，因为从cin还没有元素被读进来。一会之后就有了。

这个输入流迭代器的第二个部分就是这个结束标志，如下面我创建的这样：

```
istream_iterator<string> end;
```

标准容器使用容器末端之后一个元素的特殊值来表示容器的结束，任何算法当它碰到这个特殊值时都会停止。因为一个输入流迭代器在内存中事实上没有最后一个元素，因此它使用一个不带任何参数的构造函数来创建一个结束点的值，任何算法在这个点上都应该停止。

istream_iterator的最后一个部分的技术就是我应该如何使用它来获取值。一个方

www.TopSage.com

大学网 TopSage.com

便的方法就是使用一个容器的范围构造函数来把所有从流中来的元素都放置到一个容器中。例如，如果你构造一个带两个迭代器的矢量的话，这个构造函数将把每一个元素复制出这两个迭代器所指向的范围。如果把这个我创建的start迭代器和end迭代器传入的话，它就像如下这样做就可以：

```
vector<string> v(start, end);
```

这就是这些值从这个流中事实上读取的地方。当v被构造好时，它就从start指向的地方向前开始读取直到碰到end结束处为止。每次v都从*start中读取，它等价于从流cin中读取：

```
cin >> v[i]; // v is a vector<string>
```

换句话说，下一个从cin中读取的值被转换成一个字符串，并被插入到这个矢量中。

注意： 当你使用cin作为这个输入流时，需要你自己通过你使用的平台来决定什么是标志一个文件的结束，在这个结束处你的流应该停止。在微软的平台上，我需要使用Enter键、Ctrl+Z组合键来结束一个流的输入。你需要在你的平台上验证一下你应该使用什么，但一般也是这些键的组合。

输出流迭代器基本上和输入流迭代器相同。在示例7-11中，通过创建一个指向cout流的ostream_iterator迭代器，我把我的矢量中的值复制到cout流中，如下所示：

```
copy(v.begin(), v.end(), ostream_iterator<string>(cout, ", "));
```

这个ostream_iterator模板参数说明我将要赋给它的所有元素都是字符串。ostream_iterator的第一个构造函数参数是这个我将要向它输出的流（它可以是任意的输出流，包括ofstreams和ostringstreams），并且第二个参数就是分隔符。这就提供了把一个范围内的值输出到标准输出流中的方便办法，我常常用它来做调试。

在输出的样式上，如给带括号或花括号的序列自动换行或者避免序列末端的最后一个分隔符出现等，如果你想有更多控制的话，你就需要多费一点代码。示例7-12给出了printContainer和printRange的函数体，在本章的所有例子中，我都用到第一个。

示例7-12 实现你自己的打印函数

```
#include <iostream>
#include <string>
#include <algorithm>
#include <iterator>
#include <vector>

using namespace std;

template<typename C>
void printContainer(const C& c, char delim = ',', ostream& out = cout) {
   printRange(c.begin(), c.end(), delim, out);
}
```

www.TopSage.com

```
template<typename Fwd>
void printRange(Fwd first, Fwd last, char delim=',', ostream& out=cout) {
   out << "{";
   while (first != last) {
      out << *first;
      if (++first != last)
          out << delim << ' ';
   }
   out << "}" << endl;
}

int main() {

   cout << "Enter a series of strings: ";
   istream_iterator<string> start(cin);
   istream_iterator<string> end;
   vector<string> v(start, end);

   printContainer(v);
   printRange(v.begin(), v.end(), ';', cout);
}
```

printRange是一个更为一般的方法，因为它操作在一个范围上（在7.10节中有详细的解释），但printContainer在打印整个容器时更为方便。有很多可以这样做的办法。一个最容易想到的办法就是重定义操作符<<，让它操作在一个输出流和一个容器上，并且提供一个客户提供的算符和for_each标准算法一起使用来把每一个元素都输出到一个流上。

www.TopSage.com

第8章

类

8.0 概述

这一章包括C++类相关的几个共同问题的解决方案。本章的各节基本上是独立的，但是被组成了两个部分，每部分基本上都占到这一章的一半左右。本章的前半章包含了你曾经经历的在构造一个类的对象时几个共同问题的解决方案，如使用函数来创建对象（常常被称为工厂模式）或者使用构造函数和析构函数来管理资源。后半章包括对象构造后的几个问题的解决方案，如在运行时决定对象的类型，以及各种各样的实现技术，如通过一个抽象类来创建一个接口等。

当然，作为支持面向对象编程的语言，类是C++的一个重要的功能，通过使用类，你可以做很多不同的事情。这一章并不包含那些解释类型的基本特性的章节：虚函数（多态）、继承和封装。在进行阐述之前，我假定你已经对一个一般的面向对象的设计模式都已经熟悉，无论它是C++或者别的语言，如Java或Smalltalk。当然，本章的目的是为了给一些当你使用C++来实现面向对象设计时你可能碰到的一些呆板的难点问题提供一些解决办法。

面向对象设计和相关的一些设计模式本身就是一个大的主题，关于这个主题的文献资料可以说是汗牛充栋。在这一章中，我仅仅是提到了一部分设计模式的名字，它们都是有助于那些C++的工具中提供的一个优雅的但可能不是显而易见的解决方案。如果你还不是那么熟悉设计模式的概念，我推荐你去读一下由Gamma等编写的《设计模式》这本书（Addison Wesley），因为在软件工程中知道这些模式还是有用的；但是它并不是学习这一章的先决条件。

www.TopSage.com

8.1 初始化类成员变量

问题

如何初始化成员变量？这个成员变量是本地类型、指针或者引用类型。

解决方案

使用初始化列表来给成员变量设置初始值。示例8-1给出了一个例子来给本地类型、指针和引用类型的变量赋值。

示例 8-1 初始化类成员变量

```
#include <string>

using namespace std;

class Foo {
public:
   Foo() : counter_(0), str_(NULL) {}
   Foo(int c, string* p) :
       counter_(c), str_(p) {}
private:
   int counter_;
   string* str_;
};

int main() {

   string s = "bar";
   Foo(2, &s);
}
```

讨论

你应该总是初始化本地变量，特别当它们是类成员变量时。另一方面，类成员变量应该有一个定义好的构造函数可以用来初始化它们，所以你不用总是不得不初始化它们。让一个本地变量处在一个没有初始化的状态，将会带来很多问题。C++ 中有很多不同的办法可以用来完成这项工作，在这一节中我们将要讨论它。

最简单的初始化是本地类型。整型、字符、指针等是很容易处理的。请看如下一个简单类和它的默认构造函数：

```
class Foo {
public:
   Foo() : counter_(0), str_(NULL) {}
   Foo(int c, string* p) :
  counter_(c), str_(p) {}
private:
   int counter_;
   string* str_;
};
```

在这个构造函数中使用初始化列表来初始化成员变量，并且避免在构造函数体中进行成员变量的初始化工作。这就能让构造函数能专注于在构造过程中需要的逻辑上，并且这

www.TopSage.com

样能容易地定位成员变量的初始化。当你需要类或引用成员变量，或者你需要有效地处理异常时，使用初始化列表而不是构造函数体来初始化成员变量的优点就很明显。

注意： 成员变量初始化的顺序是根据它们在类中声明的顺序而不是根据它们在初始化列表中的顺序。

如示例 8-1 中 Foo 类中的成员变量的初始化：

```
class Foo {
public:
   Foo() : counter_(0), str_(NULL), cls_(0) {}
   Foo(int c, string* p) :
       counter_(c), str_(p), cls_(0) {}
private:
   int counter_;
   string* str_;
   SomeClass cls_;
};
```

在 Foo 类的默认构造函数中，你不需要初始化 cls_ 成员变量因为它的默认构造函数将会被调用。但如果你需要使用一个参数来构造 Foo 的话，你就应该把这个参数加到初始化列表中，就像前面我在构造函数体中给它赋值一样。通过采用初始化列表惯例，你应避免在构造 cls_ 时的额外步骤（因为如果你在构造函数体中给 cls_ 赋一个值，cls_ 首先通过它的默认构造函数构造出来，然后使用赋值操作符来给它赋值，而不是仅被构造一次），但是你能获得自动的异常处理。如果在初始化列表中一个对象被构造时，并且在构造过程中这个对象抛出了一个异常，那么运行时环境将会销毁所有其他在这个列表中已经被构造出来的对象，并且异常将会到达这个构造函数的调用者。另一方面，你如果是在构造函数体中赋值的话，你将不得不用 try/catch 代码块来处理异常。

引用则更为复杂：引用变量的初始化（和常量型成员）事实上需要使用初始化列表。根据标准，一个引用必须一直指向一个简单变量，并且不能被改变指向另一个变量。使一个引用不引用某个对象是不可能的。因此，当一个类的成员变量是一个引用类型时就必须给它赋一个有意义的值，所以它就必须在初始化时就完成，如在初始化列表中。

下面的代码行在 C++ 中是不允许的：

```
int& x;
```

也就是说，你不可能声明一个引用类型的变量但不初始化它。相反，你必须初始化它使得它指向某个对象。对于非成员变量，它的初始化如下所示：

```
int a;
int& x = a;
```

这是好的，但对于类来说，就会产生一个问题。假设你有一个带一个引用类型成员变量的类，如下：

www.TopSage.com

大家网 TopSage.com

```
class HasARef {
public:
   int& ref;
};
```

大部分编译器都可以接受它直到你创建一个这个类的实例为止，如下所示：

```
HasARef me;
```

这个时候，就会产生一个错误。下面就是gcc产生的错误：

```
error: structure 'me' with uninitialized reference members
```

相反，使用初始化列表：:

```
class HasARef {
public:
   int &ref;
   HasARef(int &aref) : ref(aref) {}
};
```

然后，当你准备创建这个类的一个实例时，你给它提供一个变量让ref引用变量引用它，如下所示：

```
int var;
HasARef me(var);
```

这就是你该如何安全地和有效地来初始化成员变量。总的来说，只要有可能都应使用初始化列表并且应避免在构造函数体中初始化成员变量，除非你必须这样做。即使你不得不在构造函数体中给这个成员变量做一些操作，你也应最少使用初始化列表来给这个成员变量设置一个合法的初始值，然后再在构造函数体中更新它。

请参考8.2节。

8.2 使用函数创建对象

问题

不要使用new操作符来在堆上创建对象，你需要使用一个函数（成员函数或独立函数）来完成对象的创建工作，这样你就可以动态地决定要创建的对象类型。这类功能行为也就是抽象工厂设计模式所实现的功能。

解决方案

你有很多种选择。你可以：

- 让这个函数在堆上创建一个对象的实例，并且返回一个指向这个对象的指针（或者用这个新创建的对象的指针来更新这个传入的指针）
- 让这个函数创建并返回一个临时对象

示例8-2说明了如何来实现以上两种做法。例子中的Session类可以是你不需要应用程序代码直接（例如使用new操作符创建）创建的任何类，但是这个创建过程则需要另一个类来管理；在这个例子中，这个具有管理功能的类是SessionFactory。

示例8-2 创建对象的函数

```
#include <iostream>

class Session {};

class SessionFactory {

public:
   Session Create();
   Session* CreatePtr();
   void Create(Session*& p);
   // ...
};

// Return a copy of a stack object
Session SessionFactory::Create() {
   Session s;
   return(s);
}

// Return a pointer to a heap object
Session* SessionFactory::CreatePtr() {
   return(new Session());
}

// Update the caller's pointer with the address
// of a new object
void SessionFactory::Create(Session*& p) {
   p = new Session();
}

static SessionFactory f; // The one factory object

int main() {
   Session* p1;
   Session* p2 = new Session();

   *p2 = f.Create();    // Just assign the object returned from Create
   p1 = f.CreatePtr(); // or return a pointer to a heap object
   f.Create(p1);       // or update the pointer with the new address
}
```

讨论

示例8-2说明了用来实现一个返回一个对象的函数的几个不同的方法。如果这个正在被分配的对象来自一个对象池，或者这个对象与硬件结合得很紧密，或者你希望这个对象的析构由别的对象而不是调用者本身来控制的话，你就会想到这样来创建对象而不是直接使用new操作符来创建。这样创建对象还有很多别的原因（这就是为什么还有一个这方面的一个设计模式）；我已经给出过了几个这样的原因。C++中实现工厂模式是直接的。

返回一个新的堆对象的地址或者使用这个新创建对象的引用来更新这个指针参数是实现这个功能最通常的做法。它们的实现在示例8-2中都有，并且都是很容易的，因此并不需

www.TopSage.com

要更多的解释。然而，不是那么常见的是从这个函数中返回整个对象，最有可能的是它带有些告诫。

通过在函数体中的栈上创建一个对象返回一个临时的对象。当这个函数返回时，编译器把这个临时对象中的数据复制到另一个事实上是由这个函数返回的临时对象中。最后，通过赋值操作符，在这个调用函数中的对象被赋给了这个临时对象的值。这就意味着，这里有两个对象被创建了：这个在工厂函数中的对象和从这个函数返回的临时对象，并且它的内容又被复制到了目标对象中。这有很多额外的副本（尽管编译器可能会把这个临时对象优化掉），因此，你一定要意识到这个过程，特别是当这个对象很大或者调用这个工厂成员函数的频率很高时。

复制一个临时对象的技术仅仅对于值对象来说比较合适，也就意味着当一个对象被复制时，这个新的对象版本等于这个原始对象。对于大部分对象来说，这样是合适的，但对于别的一些，就不是那么合适。例如，考虑创建一个侦听一个网络端口的类的对象。当你实例化这个对象时，它就开始侦听这个端口，因此你不能把这个对象复制到另一个新的对象中，因为这样的话，你就会有两个对象侦听同样的端口。在这种情况下，你愿意返回这个对象的地址。

如果你正在写一个函数或一个成员函数来创建一个对象的话，你应该看一下8.12节。通过函数模板，你可以实现一个返回一个任意类型对象的简单的函数。例如：

```
template<typename T>
T* createObject() {
   return(new T());
}

MyClass* p1 = createObject();
MyOtherClass* p2 = createObject();
// ...
```

如果你需要一个简单的能够以一种同样的方式创建任意多个类（或者一组相关的类）的对象的工厂函数，而不是多次不得不写一个冗余的工厂函数的话，这种方法是便利的。

请参考8.12节。

8.3 使用构造函数和析构函数管理资源

问题

对于代表某些资源的类，如何使用它的构造函数来获取资源并且使用析构函数来释放资源？这种技术常常称为资源获取初始化（RAII）。

解决方案

在构造函数中分配或获取资源，并且在析构函数中释放资源。这就减少了用户必须去处理异常的一些代码。

www.TopSage.com

示例 8-3 使用构造函数和析构函数

```
#include <iostream>
#include <string>

using namespace std;

class Socket {
public:
   Socket(const string& hostname) {}
};

class HttpRequest {
public:
   HttpRequest (const string& hostname) :
      sock_(new Socket(hostname)) {}
   void send(string soapMsg) {sock_ << soapMsg;}
  ~HttpRequest () {delete sock_;}
private:
   Socket* sock_;
};

void sendMyData(string soapMsg, string host) {
   HttpRequest req(host);
   req.send(soapMsg);
   // Nothing to do here, because when req goes out of scope
   // everything is cleaned up.
}

int main() {
   string s = "xml";
   sendMyData(s, "www.oreilly.com");
}
```

讨论

构造函数和析构函数的保证给编译器提供了一个很好的清理办法。典型地，在构造函数中，你初始化一个对象并给它分配任何的资源，并且在析构函数中把这些资源清理掉。这是正常的。但是程序员有个倾向就是使用创建、打开、使用和关闭这一序列的事件，在这里类的用户要求显式打开和关闭某些资源。一个文件类就是一个很好的例子。

RAII的通常的论据就是如此。示例8-3中，我容易地设计了HttpRequest类来让类的用户做更少的工作。例如：

```
class HttpRequest {
public:
   HttpRequest ();
   void open(const std::string& hostname);
   void send(std::string soapMsg);
   void close();
  ~HttpRequest ();
private:
   Socket* sock_;
};
```

通过这种方法，sendMyData函数的一个可能的版本如下所示：

```
void sendMyData(std::string soapMsg, std::string host) {
   HttpRequest req;

   try {
      req.open();
```

www.TopSage.com

```
        req.send(soapMsg);
        req.close();
    }
    catch (std::exception& e) {
        req.close();
        // Do something useful...
    }
}
```

这样使得工作多且没有任何的好处。这样的设计使得类的用户需要写更多的代码并且需要通过清理你的类来处理异常（假设在你的析构函数中不调用close函数）。

RAII有很广的应用，特别是当在你没有使用try/catch代码块的地方有异常抛出但你又想保证某些代码还是会被执行时。考虑一个桌面应用，它需要某些工作正在进行时在状态栏或标题栏中显示一个信息：

```
void MyWindow::thisTakesALongTime() {
    StatusBarMessage("Copying files...");
    // ...
}
```

StatusBarMessage类所做的就是在它被构造时在适当的窗口中更新状态信息，并且在它被析构时把这个状态信息设置回空串（或者任何以前的信息）。下面是它的关键点：如果这个函数正常返回或抛出一个异常，StatusBarMessage类都能完成它的工作。对一个超过了它的作用域的栈变量，编译器能保证析构函数被调用把它释放。不用这种方法，thisTakesALongTime函数的作者就需要自己去仔细处理这种控制逻辑，使得如果这个操作失败或用户取消这个操作等，这个错误信息不至于一直保持在窗口上。更进一步来说，比起自己来处理这种控制逻辑，直接调用这个函数将需要更少的代码和更少的错误。

如果你以前没有使用过它的话，你虽然能发现它可以在很多地方都有用，但RAII也不是万能药。另一个好的例子就是锁操作。如果你使用RAII来管理一些资源：如线程、对象池中的对象、网络连接等的锁操作的话，你将会发现这种方法有更为健壮的异常安全机制，并且需要更少的代码。事实上，这就是Boost线程库中锁操作的实现机制，它使得用户能更为清晰地编码。请参考第12章中Boost线程库的讨论。

8.4 自动添加新的类实例

问题

在不需要类的用户做任何特别处理的情况下，如何在一个简单的容器中存储一个类的所有实例？

解决方案

在一个类中包含一个静态的成员变量并且这个成员变量是一个容器，如一个定义在<list>中的链表。在构造函数中向这个容器中增加一个对象的地址并且在析构函数中把这个对象的地址删除掉。示例8-4给出了一个简单的例子：

www.TopSage.com

大家网 TopSage.com

示例 8-4 跟踪对象

```
#include <iostream>
#include <list>
#include <algorithm>

using namespace std;

class MyClass {
protected:
   int value_;
public:
   static list<MyClass*> instances_;
   MyClass(int val);
  ~MyClass();
   static void showList();
};

list<MyClass*> MyClass::instances_;

MyClass::MyClass(int val) {
   instances_.push_back(this);
   value_ = val;
}

MyClass::~MyClass() {
   list<MyClass*>::iterator p =
      find(instances_.begin(), instances_.end(), this);
   if (p != instances_.end())
      instances_.erase(p);
}

void MyClass::showList() {
   for (list<MyClass*>::iterator p = instances_.begin();
   p != instances_.end(); ++p)
 cout << (*p)->value_ << endl;
}

int main() {
   MyClass a(1);
   MyClass b(10);
   MyClass c(100);
   MyClass::showList();
}
```

示例 8-4 的输出如下：

```
1
10
100
```

讨论

示例8-4中的方法是简单的：使用一个静态的链表来存储指向对象的指针。当一个对象被创建时，把它的地址加到这个链表中；而当这个对象被销毁时，从这个链表中删除它。当然还是有很多东西需要注意的。

就像任何静态数据成员，你不得不在一个类的头文件中声明它并且在一个实现文件中定义它。示例8-4全部都在一个文件中，因此在这里它不适用，但是请记住你应该在一个实现文件中定义一个静态的变量，而不是在头文件中。更多的解释请参考 8.5 节。

www.TopSage.com

你不必使用一个静态成员。当然，你可以使用一个全局对象，但是这个设计不是自包含的。更进一步，你不得不在别的地方分配一个全局对象，在构造函数中把它传到这个MyClass中，总的来说，需要多做一点记录。

请注意像示例8-4这样共用一个全局容器，在多个线程都创建MyClass类的实例对象这种情况下是不能正常工作的。你需要使用互斥体来使得这个共享对象的访问序列化；关于这个主题和其他别的多线程技术请参考第12章。

如果你想跟踪一个类的所有实例，你可以使用工厂模式。本质上来说，这种方法意味着客户需要使用一个函数来获得一个新对象而不是使用new操作符。详细的内容请参考8.2节。

8.5 确保一个成员变量只有一个副本

问题

如何实现一个只有一个实例的成员变量，不管这个类有多少个实例？这种类型的成员变量总的来说称为静态成员变量或类变量，相对于类的实例变量来说，常常是类的每个对象都有一个这种成员变量的实例。

解决方案

使用static关键字来声明这个成员变量，然后在一个独立的源文件中初始化它（不是在它声明的头文件中），如示例8-5所示：

示例8-5 使用一个静态成员变量

```
// Static.h
class OneStatic {
public:
    int getCount() {return count;}
    OneStatic();
protected:
    static int count;
};

// Static.cpp
#include "Static.h"

int OneStatic::count = 0;

OneStatic::OneStatic() {
   count++;
}

// StaticMain.cpp
#include <iostream>
#include "static.h"

using namespace std;

int main() {
   OneStatic a;
```

www.TopSage.com

```
    OneStatic b;
    OneStatic c;

    cout << a.getCount() << endl;
    cout << b.getCount() << endl;
    cout << c.getCount() << endl;
}
```

讨论

static在C++中是用来只允许某个类型只有一个副本的方法。如果你声明一个成员变量为静态类型的话，那么这个成员变量就只有一个实例被构造，而不管这个类型有多少个对象被构造出来。同样，在一个函数中，如果你也把一个变量声明为静态类型的话，这个变量也会最多只有一个实例被构造出来，并且这个函数从一次调用到另一次调用，它的值是保持不变的。对于成员变量，为了保证这个成员变量被正确地分配你还需要做一些额外的工作。这就是为什么示例8-5中有三个文件的原因。

首先，你不得不使用static关键字来声明这个变量。这是很容易的：在这个头文件*Static.h*中类的头部加上这个关键字就可以：

```
protected:
   static int count;
```

一旦你这样做之后，你就需要在别的源文件中定义这个变量。这就是说为这个变量分配存储的值。使用类名和这个变量的名字来引用它并且给它赋值，如下所示：

```
int OneStatic::count = 0;
```

在示例8-5中，我把这个定义放在了文件*Static.cpp*中。这是你必须做的；你不应该把定义放在头文件中。如果这样做的话，包含在每个头文件中的每个实现文件都会给这个变量分配存储空间，这样你将会得到一个链接错误，或者更坏的是，在内存中这个成员变量可能会有多个实例。如果你需要的是一个静态成员变量的话，这样将和你的设想不同。

在这个主文件*StaticMain.cp*中，你就可以看到发生的事情。类OneStatic的多个实例被创建了，并且这个类的默认的构造函数每次都增加了这个静态成员变量。在*StaticMain.cpp*这个文件中输出了它的结果：

```
3
3
3
```

每次调用getCount函数都得到同样的值，尽管每次的调用都发生在这个类的不同的实例上。

8.6 在运行时确定对象的类型

问题

在运行时，如何动态地查询某个特定类的类型？

www.TopSage.com

解决方案

使用运行时类型标识（通常也叫做RTTI）来查询一个对象的地址以得到这个地址指向的对象的类型。示例8-6给出了一个简单的例子。

示例8-6 使用运行时类型标识

```
#include <iostream>
#include <typeinfo>

using namespace std;

class Base {};
class Derived : public Base {};

int main() {

   Base b, bb;
   Derived d;

   // Use typeid to test type equality
   if (typeid(b) == typeid(d)) { // No
      cout << "b and d are of the same type.\n";
   }
   if (typeid(b) == typeid(bb)) { // Yes
      cout << "b and bb are of the same type.\n";
   }
   if (typeid(d) == typeid(Derived)) { // Yes
      cout << "d is of type Derived.\n";
   }
}
```

讨论

示例8-6说明如何使用typeid操作符来决定和比较一个对象的类型。typeid带有一个表达式或者一个类型，并且返回一个type_info类型或是这个类型的子类型（实现时定义的）对象的引用。你可以使用它的返回值来测试是否相等或获取到一个代表类型名字的字符串。例如，你可以和下面一样来比较两个对象的类型：

```
if (typeid(b) == typeid(d)) {
```

如果这两个返回的type_info对象相等，就将返回真。这是因为typeid返回一个指向一个静态对象的引用，所以如果你在两个相同类型的对象上调用它的话，你将得到指向同一个对象的索引，这就是为什么这个等式能返回真的原因。

你还可以将类型本身和typeid一起使用，如下所示：

```
if (typeid(d) == typeid(Derived)) {
```

这就能让你显式地和某个类型进行测试。

大概使用typeid最通常的事情就是调试。为了输出一个类型的名字，你可以使用type_info::name，如下所示：

```
std::cout << typeid(d).name() << std::endl;
```

www.TopSage.com

当你传递一个可能改变类型的对象时，这对调试来说是一个有用的帮助。通过name函数返回一个以null结尾的字符串是实现时定义的，但你可能希望（不是依赖）在部分时间都是这个类型的名字。当然，对于C++的内在类型，这个函数也是能工作的。

不要滥用这个技术使得你的程序的逻辑基于类型信息，除非你必须这样。总的来说，如果和下面这行代码一样的逻辑的话，就会被认为是一个不好的设计：

```
If obj has a type of X, do something else, if obj has a type of Y, do
something else.
```

这种方法被认为是一个不好的设计，是因为客户的代码包含了过多地依赖于正在使用的对象的类型信息。如果你需要类型X或类型Y的对象的特殊行为的话，由于这些重复性的if/then代码，就使得你的代码变得很杂乱。面向对象和多态行为在各个地方都存在，因此你不必如此写代码。如果你需要类型特殊的逻辑，或者一个类型相关的类型家族中的动态行为的话，你应该使用同一个基类的子类和虚函数来动态调用基于不同类型的不同行为。

RTTI是会增加系统开销的，因此默认状态下编译器不会打开它。通过命令行参数你可以让你的编译器打开RTTI的功能。当然，这也不是你可以获得类型信息的唯一办法，8.7节中说明了另外的一种技术。

请参考8.7节。

8.7 确定某个对象的类是否是另一个类的子类

问题

有两个对象，如何知道它们各自的类是否来自同一个类，或者它们的继承关系或者它们是否压根就没关系？

解决方案

使用dynamic_cast操作符来获取一个类型到另一个类型的继承关系。它的结果将告诉你类之间的关系。示例8-7给出了一个简单的例子。

示例8-7 决定类关系

```
#include <iostream>
#include <typeinfo>

using namespace std;

class Base {
public:
   virtual ~Base() {} // Make this a polymorphic class
};
class Derived : public Base {
public:
```

www.TopSage.com

大家网 TopSage.com

```
    virtual ~Derived() {}
};

int main() {

    Derived d;

    // Query the type relationship
    if (dynamic_cast<Base*>(&d)) {
        cout << "Derived is a subclass of Base" << endl;
    }
    else {
        cout << "Derived is NOT a subclass of Base" << endl;
    }
}
```

讨论

使用dynamic_cast操作符来查询两个类型之间的关系。dynamic_cast带指向某个类型的一个指针或引用并且企图把它转换成它的一个派生类的指针或引用，例如，在一个类层次中进行向下变换。如果你有一个Base*型的指针并且它指向它的派生Derived对象，那么dynamic_cast<Base*>(&d)只有在d是一个继承自Base类的类型的对象时，它才返回一个派生类型的对象。如果这个转换是不可能的（因为这个Derived不是一个直接或间接地来自Base类的子类），那么这个转换将会失败并且返回NULL，如果你给dynamic_cast传递一个派生对象的指针的话。如果它是一个引用，那么一个bad_cast异常将会抛出。并且这个派生类必须是使用public继承基类的，且这个继承是没有二义性的。这个结果将告诉你一个类是另外一个类的子类。示例8-7的使用如下：

```
if (dynamic_cast<Base*>(&d)) {
```

以上将返回一个非空的指针，因为d是一个继承自Base基类的类的对象。在任意的一对类上都能查询出它们之间的关系。唯一的要求就是这个对象参数是一个多态性的类型，这也就是说它最少都有一个虚函数。如果没有的话，它将不能编译成功。这就保证了不会产生很多令人头痛的问题，因为不带虚函数的类继承是不太可能的。

如果这个语法形式令你觉得难懂的话，你可以使用一个宏来重新定义它：

```
#define IS_DERIVED_FROM(baseClass, x) (dynamic_cast<baseClass*>(&(x)))
//...
if (IS_DERIVED_FROM(Base, l)) { // ...
```

这个类型信息是不能随便改变的，因为dynamic_cast必须在运行时转换类的层次关系以决定一个类是否是另一个类的子类，所以你必须对它能使用在什么地方有所了解。另外，默认状态下，编译器并不包含这类信息因为它需要用到RTTI，而RTTI是会带来系统开销的，所以并不是每个人都应该使用这个特性，如果你需要用的话就把编译器的开关打开。

请参考8.6节。

www.TopSage.com

8.8 为类的每个实例指定一个唯一的标识符

问题

如何给类的每一个对象指定一个唯一的标识符？

解决方案

使用一个静态的成员变量来跟踪下一个可以使用的标识符。在构造函数中，给当前对象赋一个下次可以使用的值并且增加这个静态成员变量。如示例8-8中给出了一个简单的例子。

示例 8-8 赋一个唯一的标识符

```
#include <iostream>

class UniqueID {
protected:
   static int nextID;
public:
   int id;
   UniqueID();
   UniqueID(const UniqueID& orig);
   UniqueID& operator=(const UniqueID& orig);
};

int UniqueID::nextID = 0;

UniqueID::UniqueID() {
   id = ++nextID;
}

UniqueID::UniqueID(const UniqueID& orig) {
   id = orig.id;
}

UniqueID& UniqueID::operator=(const UniqueID& orig) {
   id = orig.id;
   return(*this);
}

int main() {
   UniqueID a;
   std::cout << a.id << std::endl;
   UniqueID b;
   std::cout << b.id << std::endl;
   UniqueID c;
   std::cout << c.id << std::endl;
}
```

讨论

使用一个静态成员变量来跟踪下一个可以使用的标识符。在示例8-8中，我使用了一个静态的整型变量，但是你可以使用任意类型，只要对于这个唯一标识符你可以使用一个函数来产生它的唯一值就可以。

在这种情况下，这个标识符的值是不会重复使用的，直到达到了一个整型数所能表达的

www.TopSage.com

最大值。一旦你删除了一个对象，那么这个对象的标识符的唯一值就也跟着消失，直到重新启动这个程序或标识符的值达到了它的最大值并且继续重复使用。这个程序中标识符的唯一性有很多有趣的优点。例如，如果你正使用一个内存管理的类库，它能搅乱内存或使一些指针无效，通过这个唯一值你就能保证它在每个对象中都不同。在8.4节中，如果你使用唯一值，并且使用map而不是list，你就能用这个唯一的标识符来定位你的对象。为了达到这个目标，你可以简单地用一个唯一的ID来映射你的对象，如下所示：

```
static map<int, MyClass*> instmap;
```

通过这种方法，任何跟踪了一个对象的标识符的代码就在不持有这个对象的索引的前提下都能在任何地方找到这个对象。

但这并不是它的所有功能。考虑如下的情况，当你需要在一个标准容器（矢量、链表、集合等）中增加一个这种对象时。这个标准容器包含你增加的对象的一个副本，而不是这个对象的索引或是指针（除非这个容器是一个指针类容器）。因此，标准容器希望它包含的对象如同是一个值对象，这就是说当你使用赋值操作符或者复制构造函数来给它赋值时，一个等于这个对象的新对象就会产生。

这就是说你必须对你的唯一对象的行为有所了解。当你使用一个唯一的标识符来创建一个对象并把它加到一个容器中时，你将有两个对象具有相同的标识符，除非在你赋值操作中做出一些特别的处理。在你赋值操作和复制构造操作中，你需要处理这个唯一值以使它具有意义。容器中的对象等于原来的那个对象是否有意义？如果是，那么标准的赋值操作和复制构造操作就可以使用，但是你也应显式指出来，这样你的类的用户就能知道你这样做的目的并且就不会仅仅只是忘记容器的工作方式而变幸运。例如，在你的复制构造操作和赋值操作中使用这个相同的标识符，你可以如下这样做：

```
UniqueID::UniqueID(const UniqueID& orig) {
   id = orig.id;
}

UniqueID& UniqueID::operator=(const UniqueID& orig) {
   id = orig.id;
   return(*this);
}
```

但是基于你的应用程序的上下文，在你的容器中，给你的对象创建另一个唯一值可能更有意义。在这种情况下，你就可以像在通常的构造函数中那样再一次使用这个静态变量，如下所示：

```
UniqueID::UniqueID(const UniqueID& orig) {
   id = ++nextID;
}

UniqueID& UniqueID::operator=(const UniqueID& orig) {
   id = ++nextID;
   return(*this);
}
```

www.TopSage.com

尽管你可能还不是很清楚。如果你在一个多线程的环境中使用UniqueID的话，你仍然会有麻烦，因为这个静态变量的访问是不同步的。更多的关于多线程中资源的使用信息，请参考第12章。

请参考8.3节。

8.9 创建单个类

问题

如何实现一个只能被实例化一次的类，并且需要提供一个方法来使你的客户每次访问这个类都返回这个同样的单一对象？这通常称之为单例模式或者单一类。

解决方案

创建一个静态的成员变量并且这个成员变量是一个指向当前类的指针，通过使用private私有修饰符来限制构造函数的使用来创建这个类的对象，并且提供一个公有的静态成员函数，客户能够通过这个成员函数来访问这个单一的静态实例。示例8-9说明是如何这样做的。

示例8-9 创建单个类

```
#include <iostream>

using namespace std;

class Singleton {
public:
   // This is how clients can access the single instance
   static Singleton* getInstance();

   void setValue(int val) {value_ = val;}
   int  getValue() {return(value_);}

protected:
   int value_;

private:
   static Singleton* inst_;   // The one, single instance
   Singleton() : value_(0) {} // private constructor
   Singleton(const Singleton&);
   Singleton& operator=(const Singleton&);
};

// Define the static Singleton pointer
Singleton* Singleton::inst_ = NULL;

Singleton* Singleton::getInstance() {
   if (inst_ == NULL) {
      inst_ = new Singleton();
   }
   return(inst_);
}

int main() {
```

www.TopSage.com

```
    Singleton* p1 = Singleton::getInstance();
    p1->setValue(10);
    Singleton* p2 = Singleton::getInstance();
    cout << "Value = " << p2->getValue() << '\n';
}
```

讨论

有很多情况都需要你的类最多只有一个实例。这就是为什么单类是一种设计模式。通过简单的几步，在C++中很容易实现单个类。

当你决定只需要某个事物的一个实例时，你应该首先想到static关键字。就像在8.5节中我描述的那样，一个静态成员变量就是在内存中最多只有一个实例的事物。使用静态成员变量来跟踪你的单个类的那个对象，就像我在示例8-9中描述的那样：

```
private:
    static Singleton* inst_;
```

使用私有private修饰符能保证客户代码不知道这个成员变量。但在实现文件中你必须把这个静态的成员变量初始化成NULL：

```
Singleton* Singleton::inst_ = NULL;
```

为了阻止客户代码实例化这个类，必须使得构造函数也是私有的，特别是默认的构造函数。

```
private:
    Singleton() {}
```

这样，任何企图在堆或栈上创建这个类的实例，编译器都能友好地给出一个编译错误。

现在你已经创建了一个静态变量来跟踪一个单例对象，并且你通过限制它们的构造函数从而禁止了单例对象的创建，余下的就是提供一个客户能够访问这个单例对象实例的方法。如下使用一个静态成员函数来实现这个目标：

```
Singleton* Singleton::getInstance() {
    if (inst_ == NULL) {
        inst_ = new Singleton();
    }
    return(inst_);
}
```

你能够看出它是如何工作的。如果这个静态的单例指针是NULL的话，就创建这个对象。如果它已经被创建的话，就返回它的地址。通过调用这个静态成员函数，客户代码就能访问这个单例对象实例。

```
Singleton* p1 = Singleton::getInstance();
```

并且如果你不期望你的客户代码处理指针的话，你也可以返回一个引用，如下所示：

www.TopSage.com

大家网 TopSage.com

```
Singleton& Singleton::getInstance() {
   if (inst_ == NULL) {
      inst_ = new Singleton();
   }
   return(*inst_);
}
```

这个指针有两方面的作用，一方面它阻止了客户代码创建这个单例对象的实例；另一方面，它提供了一个客户代码能够访问的接口。

请参考 8.3 节。

8.10 创建含抽象基类的接口

问题

如何定义一个子类将来可以实现的接口，但是从概念上来说，接口的定义是抽象的，并且是不可以实例化的？

解决方案

通过声明最少一个纯虚函数来创建一个抽象类从而定义这个接口。客户代码能够用不同的实现来继承这个抽象类从而保证了这个共同的接口。示例8-10说明了如何定义一个读配置文件数据的抽象类。

示例 8-10 使用一个抽象基类

```
#include <iostream>
#include <string>
#include <fstream>

using namespace std;

class AbstractConfigFile {

public:
   virtual ~AbstractConfigFile() {}

   virtual void getKey(const string& header,
                       const string& key,
                             string& val) const = 0;
   virtual void exists(const string& header,
                        const string& key,
                              string& val) const = 0;
};

class TXTConfigFile : public AbstractConfigFile {

public:
            TXTConfigFile() : in_(NULL) {}
            TXTConfigFile(istream& in) : in_(&in) {}
   virtual ~TXTConfigFile() {}

   virtual void getKey(const string& header,
                       const string& key,
                             string& val) const {}
   virtual void exists(const string& header,
```

www.TopSage.com

```
                        const string& key,
                              string& val) const {}

protected:
   istream* in_;
};

class MyAppClass {
public:
   MyAppClass() : config_(NULL) {}
  ~MyAppClass() {}
   void setConfigObj(const AbstractConfigFile* p) {config_ = p;}
   void myMethod();

private:
   const AbstractConfigFile* config_;
};

void MyAppClass::myMethod() {

   string val;
   config_->getKey("Foo", "Bar", val);
   // ...
}

int main() {


   ifstream in("foo.txt");
   TXTConfigFile cfg(in);

   MyAppClass m;

   m.setConfigObj(&cfg);

   m.myMethod();
}
```

讨论

一个抽象基类（常常用ABC来称呼）是一个不能被实例化的类，因此它也就只是起到一个接口的作用。如果一个类声明了最少一个纯虚函数或者它继承了一个纯虚函数但没有它的实现的话，那么这个类就是抽象类。因此，如果这个ABC的一个子类需要被实例化的话，它就不得不实现ABC类中所有的虚函数，也就是说这个类支持由ABC声明的这个接口。

一个继承自ABC的子类（实现了它的所有纯虚函数）遵守了由这个接口定义的契约。考虑示例8-10中的MyAppClass和TXTConfigFile类。MyAppClass类中有一个指针成员，它指向这个抽象类AbstractConfigFile的一个对象：

```
const AbstractConfigFile* config_;
```

（我声明它为常量型是因为MyAppClass不应该改变这个配置文件，它仅只是从这个文件中读数据）。用户可以调用设置成员函数setConfigObj来给这个MyAppClass设置一个配置文件。

当给MyAppClass类使用这个配置文件时，就像 MyAppClass::myMethod方法那

www.TopSage.com

样，它就能调用声明在AbstractConfigFile类中的任何函数而不用管这个正在被使用的配置文件是什么类型。它可以是TXTConfigFile、XMLConfigFile或者任何别的继承自 AbstractConfigFile 的子类。

总的来说，多态就是继承的一个优点：如果你的代码引用了一个基类对象，在它上面调用了一个虚函数，它将动态调用这个基类的子类的正确版本，只要这个对象事实上是引用了子类的对象就可以。但是如果它的基类不是 ABC 的话，又会如何呢？

这有两个区别。一个纯接口类（也就是ABC没有提供任何实现）的作用就是给它的子类提供了一个子类实例化时必须遵守的契约。常常这就意味着这个纯接口类的子类上的is-a测试可能不会成功（意味着你不可以说一个子类的对象是这个基类的一个对象），但是这个行为相似的测试可以成功。这就允许你对于它是什么和它能做什么有一定的分离。想象一下一个超人。他是一个人，但是他也是超级英雄。超级英雄可以像鸟一样飞，但是你不能说超级英雄就是一只鸟。你可以设计一个继承自超级英雄的类，就像我在示例 8-11 中做的那样。

示例 8-11 使用一个纯接口

```
class Person {
public:
   virtual void eat()   = 0;
   virtual void sleep() = 0;
   virtual void walk()  = 0;
   virtual void jump()  = 0;
};

class IAirborne {
public:
   virtual void fly()  = 0;
   virtual void up()   = 0;
   virtual void down() = 0;
};

class Superhero : public Person,     // A superhero *is* a person
                  public IAirborne { // and flies
public:
   virtual void eat();
   virtual void sleep();
   virtual void walk();
   virtual void jump();
   virtual void fly();
   virtual void up();
   virtual void down();
   virtual ~Superhero();
};

void Superhero::fly() {
   // ...
}

// Implement all of the pure virtuals in Superhero's superclasses...

int main() {

   Superhero superman;
   superman.walk(); // Superman can walk like a person
```

www.TopSage.com

```
   superman.fly();  // or fly like a bird
}
```

很多不同事物都可以飞，但是你不能调用一个接口，例如，IBird和IAirborne指示任何支持它们的都是可以飞的。所有它们能做的只是确保能和一个继承自IAirborne的对象一起工作，也就是客户代码能够调用fly、up和down函数。

这第二个区别就是ABC只是定义了一个抽象的实体，并且作为对象它是没有任何意义的，因为总的来说它只是为继承使用的。在这种情况下，那个is-a测试只是支持继承的，但是ABC本身是抽象的，它本身是没有任何实现的，作为对象它是不可以实例化的。考虑示例8-10中的AbstractConfigFile类：实例化一个AbstractConfigFile有任何意义吗？没有，只有在实例化一个代表具体的配置文件时，它才有意义。

下面是关于抽象类和纯虚函数的一些规则。只有在满足以下一些条件时一个类才是抽象的：

- 它声明了至少一个纯虚函数
- 它继承了最少一个纯虚函数，但是这个虚函数没有任何实现

一个抽象类是不能实例化的。然而通过一个抽象类你可以：

- 具有数据成员
- 具有非虚函数
- 给纯虚函数提供实现
- 做一个通常的类可以做的大部分事情

换句话说，除了不能实例化它之外，别的通常的类能做的事情抽象类也能做。

在实现的时候，C++中使用ABC基类需要一定的判断。无论是否把ABC当成一个纯接口，这都取决于你自己。例如，在某个时候，在这个超级英雄的例子中，假设我决定这个Person类应该是抽象的，但由于每个人都有一个名字和姓，我又想把这些成员加上并且相应地把设置和获取成员函数也加上，这样这个子类的作者就不用自己来加上这些：

```
class Person {
public:
   virtual void   eat()   = 0;
   virtual void   sleep() = 0;
   virtual void   walk()  = 0;
   virtual void   jump()  = 0;
   virtual void   setFirstName(const string& s) {firstName_ = s;}
   virtual void   setLastName(const string& s) {lastName_ = s;}
   virtual string getFirstName() {return(firstName_);}
   virtual string getLastName() {return(lastName_);}
protected:
   string firstName_;
   string lastName_;
};
```

www.TopSage.com

现在，如果这个超级英雄子类想改写其中的某个函数的话，它能做的就是使用基类的名字来修饰这个函数，从而在子类的函数中就能调用这个基类中的函数版本。例如：

```
string Superhero::getLastName() {
   return(Person::getLastName() + " (Superhero)");
}
```

偶尔，你还是需要使得这个函数变成纯虚函数并提供一个默认的实现。你可以在语法上紧跟在它的声明后使用=0来使得它成为一个纯虚函数，并且在别的地方提供一个它的定义，如下所示：

```
class Person {
// ...
virtual void   setFirstName(const string& s) = 0;
// ...
Person::setFirstName(const string& s) {
   firstName_ = s;
}
```

通过这样做，你就能迫使子类重写这个虚函数，但是你加上类名来限制它的话，你还是可以来调用这个默认的版本。

最后，如果在你的基类中提供了虚的析构函数，你需要给它提供一个函数体。这是因为子类的析构函数需要自动调用基类的析构函数。

8.11 编写一个类模板

问题

如何实现一个类，它的成员在不同的情况下要求是不同的类型，并且使用传统的多态行为是笨重的和冗余的？换句话说，作为类的设计者，当类的用户实例化一个类时，你期望他能够选择你的类的各个部分的类型，而不是在类的原始定义中设置它们。

解决方案

使用类模板来参数化类型，这些参数化类型可以被使用来声明类成员（或者更多）。这就是说，使用类型占位符来写你的类；因此，让类模板的用户来选择使用的类型。请参考示例8-12中的TreeNode类模板例子，它可以被使用来指向任何类型。

示例8-12 写一个类模板

```
#include <iostream>
#include <string>

using namespace std;

template<typename T>
class TreeNode {
public:
   TreeNode(const T& val) : val_(val), left_(NULL), right_(NULL) {}
  ~TreeNode() {
      delete left_;
```

www.TopSage.com

大家网 TopSage.com

```
      delete right_;
   }

   const T& getVal() const {return(val_);}
   void setVal(const T& val) {val_ = val;}
   void addChild(TreeNode<T>* p) {
      const T& other = p->getVal();
      if (other > val_)
         if (right_)
            right_->addChild(p);
         else
            right_ = p;
      else
         if (left_)
            left_->addChild(p);
         else
            left_ = p;
   }
   const TreeNode<T>* getLeft() {return(left_);}
   const TreeNode<T>* getRight() {return(right_);}

private:
   T val_;
   TreeNode<T>* left_;
   TreeNode<T>* right_;
};

int main() {

   TreeNode<string> node1("frank");
   TreeNode<string> node2("larry");
   TreeNode<string> node3("bill");

   node1.addChild(&node2);
   node1.addChild(&node3);
}
```

讨论

类模板给类的设计者参数化类型提供了一种方法，因此，它们在类被实例化时可被类的用户提供。尽管模板使用起来还是有点令人费解，因此，在你明白它是如何工作的之前，让我来说明一下这个例子。

考虑一下示例 8-12 中的 TreeNode 类模板的声明：

```
template<typename T>
class TreeNode { //...
```

这个template<typename T>部分是使得它是一个类模板而不是一个普通的类。这行代码的意思是说 T 是一个类型的名字，但这个类型是这个类被使用时而不是现在它被声明时指定的类型。这个参数 T 可以在 TreeNode 的整个声明和定义期间使用，就好像它是任何其他类型、本地类型或者用户自定义类型。例如，我有一个私有成员叫做 val_，并且我想把它声明成类型 T，因此我可以声明如下：

```
T val_;
```

这就简单地声明了一个名为 val_ 的类成员，它将在后面如同我把它声明为 int、float、MyClass 或者是字符串类型一样决定它的类型。在这一点上，你可以把它想象成一个宏（例如，使用 #define），尽管它与宏的相像性也就这么多。

www.TopSage.com

你的类型参数可以与你使用任何的普通类型参数的任何方式一样使用：返回值、指针、成员函数参数等等。考虑下面有关val_的getter和setter方法：

```
const T& getVal() const {return(val_);}
void setVal(const T& val) {val_ = val;}
```

getVal返回一个指向val_的常量引用，它的类型是T，setVal带一个指向T的引用并且设置val_等于它。当你碰到getLeft和getRight成员函数，事情好像有点复杂，因此我一会再回到这个主题上。稍微等待一下。

既然TreeNode被声明成带一个类型占位符，那么一些客户代码就不得不使用它。如下所示：

TreeNode是一个二叉树的简单实现。为了创建一个存储字符串值的树，你可以如下这样创建你的节点：

```
TreeNode<string> node1("frank");
TreeNode<string> node2("larry");
TreeNode<string> node3("bill");
```

这个在尖括号里的类型是当这个类模板实例化时用来代替类型T的。模板实例化是当T是一个字符串时编译器建立TreeNode的版本时所经历的过程。当它实例化时（并且只有在它实例化时），一个二进制的物理上代表TreeNode<string>才会被创建。你所得到的是一个内存的布局，这个内存布局与在你没有使用template关键字和类型参数来实现TreeNode，并且在任何你使用T的地方使用字符串代替它的内存布局相同。

对于一个给定类型参数的模板的实例化与一个类的对象的实例化类似。最主要的不同是模板的实例化发生在编译时，而对象的实例化则发生在运行时。这就意味着，如果你想使用你的二叉树来存储整型数而不是字符串的话，你需要如下这样来声明它：

```
TreeNode<int> intNode1(7);
TreeNode<int> intNode2(11);
TreeNode<int> intNode3(13);
```

就像字符串版本的二叉树那样，为这个使用整型作为它的内部类型TreeNode类模板也创建了一个二叉树实体。

一分钟以前，我说过我会重新说明这个getLeft和getRight成员函数。现在你已熟悉了模板的实例化（如果你以前不熟悉的话），这两个getLeft和getRight成员函数的声明和定义或许就更有意义：

```
const TreeNode<T>* getLeft() {return(left_);}
const TreeNode<T>* getRight() {return(right_);}
```

它们的意思是这每一个成员函数都返回一个指向T类型的TreeNode实例的指针。因此，当TreeNode给某个类型（例如字符串）实例化时，getLeft和getRight的实例化如下：

www.TopSage.com

```
const TreeNode<string>* getLeft() {return(left_);}
const TreeNode<string>* getRight() {return(right_);}
```

当然你也没有被限制只能使用一个模板参数。如果你愿意，你可以使用很多模板参数。假设你想要跟踪一个给定节点下的子节点的个数，但是你的类的用户或许在内存空间上的限制而如果能够使用短整型的话就不想使用整型。相同地，他们或许想提供别的什么而不是一个简单的内置类型来记录这个节点的使用，就像他们自己的数字类型。在任何情况下，你都能让他们这样与另外的模板参数一起使用：

```
template<typename T, typename N = short>
class TreeNode {
// ...
   N getNumChildren();

private:
   TreeNode() {}
   T val_;
   N numChildren_;
// ...
```

这样使用你的类的人就可以提供整型、短整型或者别的任何类型来跟踪每个节点的子树的大小。

当然你也可以给这个模板参数提供默认的参数，就像我在这个例子中做的那样，使用同样的语法你将可以声明一个默认的函数参数：

```
template<typename T, typename N = short>
```

和这个默认的函数参数值一起，你可以仅给某个给定的参数提供实参，如果它是最后的那个参数或者是有默认值的参数右边的每一个参数。

示例8-12中，这个模板的定义和声明都在同一个地方。常常，我这样做是为了在例子代码中节省空间，但是，在这个例子中，还有另外一个原因。模板（类或函数——请参考8.12节）仅仅是在它们被实例化时才被编译成二进制形式。因此，在一个头文件中你不能有模板的声明并且它的实现也不能在一个源文件中（例如，*cpp*文件）。其原因是没有任何东西会被编译！但是也有例外，总的来说，如果你正在写一个类模板，你应该把它的实现放到头文件或一个被这个头文件包含的内嵌文件中。

如果你这样做的话，你将需要使用一个不太熟悉的语法。就像你在一个一般的类上声明成员函数和类模板的其他部分一样，但是当你需要定义这个成员函数时，你就不得不包含一些额外的标记来告诉编译器这是给类模板的。例如，你需要定义`getVal`成员函数，如下所示（比较一下示例8-12）：

```
template<typename T>
const T& TreeNode<T>::getVal() const {
   return(val_);
}
```

函数体是相同的。

www.TopSage.com

但是对于模板，你还是应该小心，因为如果你写一个可以用在任意地方的模板，你就会使得代码膨胀，也就可能把同样参数的同样模板（TreeNode<int, short>）编译到不同的对象文件中。本质上，在不同的文件中就会存在一个实例化模板的同样的二进制代表，并且这样就会使得你的类库或者可执行文件更大。

一个避免这种情况的办法就是使用显式实例化，它是一个告诉编译器它需要给某个特殊集合的模板参数实例化一个类模板版本的方法。如果你在一个地方这样做，而这个地方将会被链接到多个客户代码中，你就能避免代码膨胀。例如，如果我知道我的应用程序的各处都需要使用TreeNode<string>，在一个共同的源代码文件中我就可以加入一行代码，如下所示：

```
// common.cpp
template class TreeNode<string>;
```

使用这个文件来建立共享类库并且那些使用这个TreeNode<string>的代码就可以动态地使用这个类库而不必包含这个类库的编译版本。其他代码就可以为这个类模板包含这个头文件，然后连接这个类库并且这样就可以避免需要它自己的副本。尽管这样做需要一些试验，因为并不是所有的编译器都会有同样的这个问题使得代码膨胀到同样的程度，但这是你可以用来减少代码的一般方法。

C++模板（类和函数）是一个非常大的主题，并且有很多专用的技术可以用来设计强大的高效模板。类模板的一个伟大的应用例子就是标准类库容器，例如，矢量、链表、集合等等，这些都是第15章的主题。出现在C++文献中的大部分有趣的开发都使用了模板。如果你对这些主题感兴趣，你应该去查阅一下新闻组 *comp.lang.std.c++* 和 *comp.lang.c++*。在这里能发现一些有趣的问题和答案。

请参考8.12节。

8.12 编写一个成员函数模板

问题

如何实现一个简单成员函数，这个成员函数可以带一个任意类型的参数，并且你不能或者不想把这个成员函数的参数限制成某个特殊类型或某几种类型（通过使用一个指向基类的指针）？

解决方案

使用成员函数模板并且把这个模板的参数声明成这个成员函数的参数的对象类型。示例8-13给出了一个简单的例子。

www.TopSage.com

示例8-13 使用一个成员函数模板

```
class ObjectManager {
public:
   template<typename T>
   T* gimmeAnObject();

   template<typename T>
   void gimmeAnObject(T*& p);
};

template<typename T>
T* ObjectManager::gimmeAnObject() {
   return(new T);
}

template<typename T>
void ObjectManager::gimmeAnObject(T*& p) {
   p = new T;
}

class X { /* ... */ };
class Y { /* ... */ };

int main( ) {
   ObjectManager om;

   X* p1 = om.gimmeAnObject<X>(); // You have to specify the template
   Y* p2 = om.gimmeAnObject<Y>(); // parameter

   om.gimmeAnObject(p1);  // Not here, though, since the compiler can
   om.gimmeAnObject(p2);  // deduce T from the arguments
}
```

讨论

当谈到函数或者类模板时，这个参数所代表的含义都有一点模糊。这有两类：模板和函数。模板参数是在这个尖括号里的参数，如示例8-13中的T，而函数参数则是传统意义上的参数。

考虑示例8-13中的类ObjectManager。它是8.2节中讨论的工厂模式的简单版本，因此我把成员函数gimmeAnObject定义成可以创建新对象的函数，用户可以使用它而不用直接使用new操作符来创建新对象。通过返回一个指向一个新对象的指针或修改这个由用户代码传递进来的指针就可以达到这个目的。让我们看看这两种方法的情形。

一个模板成员函数的声明需要你提供template关键字和模板参数。

```
template<typename T>
T* gimmeAnObject();

template<typename T>
void gimmeAnObject(T*& p);
```

这两个成员函数都是使用T作为模板参数，但是它们不是非得如此；它们仅仅是代表这个成员函数的模板参数而已，因此这个名字是无所谓的。对于你自己的模板成员函数定义，你需要做同样的事情，例如，使用template关键字和列出所有的模板参数。下面是我的定义：

www.TopSage.com

大家网 TopSage.com

```
template<typename T>
T* ObjectManager::gimmeAnObject() {
   return(new T);
}

template<typename T>
void ObjectManager::gimmeAnObject(T*& p) {
   p = new T;
}
```

有很多种办法可以用来调用模板成员函数。首先，你可以显式使用模板参数来调用模板成员函数，如下所示：

```
X* p1 = om.gimmeAnObject<X>();
```

X 仅仅是某个类的名字。或者，你可以传递一个模板参数类型的参数给这个模板成员函数来让编译器自己推导这个模板参数。例如，你可以不使用尖括号来传递任何东西以调用 gimmeAnObject 函数，如下所示：

```
om.gimmeAnObject(p1);
```

这是因为编译器通过查验 p1 进行计算并且能计算出它是一个 X*。模板推导仅能在函数模板上能工作并且只能通过函数参数才能推导出模板参数的类型。

成员函数模板在 C++ 中不是那么常用，但是它们也是很方便的，因此知道如何写一个成员函数模板也是很好的。当我想让一个成员函数与一个跟继承不相关的类型一起工作时，我就常常想到使用这个成员函数模板。例如，如果我有一个成员函数 foo，并且这个 foo 成员函数需要带一个简单的参数，而这个参数又是一个继承自某个类的类，我就不需要使用模板：我只需要把这个参数类型设置成这个基类的指针或引用。然后，这个参数类的任何子类对象就都能在这个函数中运行正常 —— 这就是 C++ 的方法。

但是你可能需要这个函数操作的参数并不是一直继承自某个同样的基类。在这种情况下，你可以把同样的成员函数多写几遍，每一次都是用一个不同的类型作参数，或者你就可以使用模板成员函数。使用模板也允许特殊化，也就给某个特定的模板参数提供一个模板的实现。但这超过了本节的讨论范围，因此对这个主题，这里不进一步讨论，但是它是一个功能强大的技术，因此如果模板编程出现在你的面前的话，我建议你好好检验一下。

请参考 8.11 节。

8.13 重载自增和自减运算符

问题

有一个类，并且在这个类中希望熟悉的自增和自减操作符有意义，那么，如何重载操作符 ++ 和操作符 --，以使得类的对象在增加和减小上更容易并且对用户更直观？

解决方案

你可以根据你的需要来重载前缀和后缀形式的++和--。示例8-14说明了一个传统重载自增和自减操作符的技术。

示例8-14 重载自增和自减操作符

```
#include <iostream>

using namespace std;

class Score {
public:
   Score() : score_(0) {}
   Score(int i) : score_(i) {}

   Score& operator++() { // prefix
      ++score_;
      return(*this);
   }
   const Score operator++(int) { // postfix
      Score tmp(*this);
      ++(*this); // Take advantage of the prefix operator
      return(tmp);
   }
   Score& operator--() {
      --score_;
      return(*this);
   }
   const Score operator--(int x) {
      Score tmp(*this);
      --(*this);
      return(tmp);
   }
   int getScore() const {return(score_);}

private:
   int score_;
};

int main() {
   Score player1(50);

   player1++;
   ++player1; // score_ = 52
   cout << "Score = " << player1.getScore() << '\n';
   (--player1)--; // score_ = 50
   cout << "Score = " << player1.getScore() << '\n';
}
```

讨论

这个自增和自减操作符对于那些代表某种整型值的类来说才有意义。它们都很容易使用，只要你了解前缀和后缀形式的自增或自减的区别并且遵循返回值的传统就可以。

考虑一下增加一个整型数。例如，某个整型变量i，有两种方法可以和它一起使用++操作符：

```
i++; // postfix
++i; // prefix
```

www.TopSage.com

它们都增加了 i 的值：第一种方法为 i 创建了一个临时副本再增加 i 的值，然后返回这个临时副本的值，而第二种方法只是增加 i 的值并返回它的值。C++允许操作符重载，也就意味着你可以使用你定义的类型（一个类或枚举）像整型那样来操作。

重载操作符++和操作符--来实现你的需要。示例8-14说明了如何重载前缀自增和后缀自增两个版本：

```
Score& operator++() { // prefix
   ++score_;
   return(*this);
}
const Score operator++(int) { // postfix
   Score tmp(*this);
   ++(*this);
   return(tmp);
}
```

前缀自增和你期望的相同，但是对于编译器来说，区别这两种方法的是一个作为后缀自增操作符声明的一部分的整型参数。它并不具有语意上的作用；在运行时，它总是以零传入的，因此你可以忽略它。

一旦你这样做的话，你可以像使用一个整型一样使用 Score 类：

```
Score player1(50);

player1++;
++player1; // score_ = 52
```

你大概注意到了操作符++的前缀自增版本的函数签名返回一个指向当前类的引用。你应该这样做（而不是返回一个无类型）因此这个正在自增或自减的对象可以使用到别的表达式中。请看来自这个例子中的下面一行代码：

```
(--player1)--;
```

很奇怪，是吗？但它证明了一点。如果这个前缀操作符--不返回任何有意义的东西，那么这个表达式就不能编译通过。下面是一个函数调用的例子：

```
foo(--player1);
```

这个 foo 期望一个 Score 类型的参数，并且这正是为了编译你希望从前缀操作符--返回的类型。

操作符重载是一个强大的功能，它能让你把那些应用到内置类型的操作应用到用户自定义类型上。其他的一些不允许操作符重载的语言的支持者认为为了各种客户定义的行为，许多的操作符都可能被重载，无可否认地会带来混乱并变得复杂。但对于简单的自增和自减操作，能够如你希望的那样来自定义你的类行为还是很好的。

请参考 8.14 节。

www.TopSage.com

8.14 为直观类行为重载算术和赋值操作符

问题

有一个类，在这个类中，一些C++的一元和二元操作符是意义的，并且期望你的类的用户能够在你的类的对象上使用这些操作符，如何实现这一点？例如，如果你有一个名为Balance的类，本质上，它包含一个浮点数值（例如，一个账户的余额），如果你可以在这个Balance类的对象上使用一些C++标准操作符的话，将会非常方便，如下所示：

```
Balance checking(50.0), savings(100.0);

checking += 12.0;
Balance total = checking + savings;
```

解决方案

作为成员函数和独立函数重载这些你需要使用的操作符，让这些各种类型的参数对于这些给定的操作符都有意义，如示例8-15所示。

示例8-15 重载一元和二元操作符

```
#include <iostream>

using namespace std;

class Balance {
   // These have to see private data
   friend const Balance operator+(const Balance& lhs, const Balance& rhs);
   friend const Balance operator+(double lhs, const Balance& rhs);
   friend const Balance operator+(const Balance& lhs, double rhs);

public:
   Balance() : val_(0.0) {}
   Balance(double val) : val_(val) {}
  ~Balance() {}

   // Unary operators
   Balance& operator+=(const Balance& other) {
      val_ += other.val_;
      return(*this);
   }
   Balance& operator+=(double other) {
      val_ += other;
      return(*this);
   }

   double getVal() const {return(val_);}

private:
   double val_;
};

// Binary operators
const Balance operator+(const Balance& lhs, const Balance& rhs) {
   Balance tmp(lhs.val_ + rhs.val_);
   return(tmp);
}

const Balance operator+(double lhs, const Balance& rhs) {
   Balance tmp(lhs + rhs.val_);
   return(tmp);
}
```

www.TopSage.com

```
const Balance operator+(const Balance& lhs, double rhs) {
   Balance tmp(lhs.val_ + rhs);
   return(tmp);
}

int main() {

   Balance checking(500.00), savings(23.91);

   checking += 50;
   Balance total = checking + savings;

   cout << "Checking balance: " << checking.getVal() << '\n';
   cout << "Total balance: "    << total.getVal() << '\n';
}
```

讨论

操作符重载最通常的情况是赋值和算术运算。有很多种类型的类中算术和赋值操作符（加、乘、取模、左/右位移）都有意义，无论你是作为数学意义还是别的什么来使用。示例8-15说明了一些重载这些操作符的一些基本的技术。

让我们以最经常重载的操作符（赋值操作符）开始讨论吧。赋值操作符是当你把一个对象赋给另一个对象时就需要使用的操作符，如下这行代码：

```
Balance x(0), y(32);
x = y;
```

第二行代码就是调用Balance::operator=(y)的一个快速的方法。赋值操作符与大部分其他的操作符都不同，因为如果你不提供它的话，一个默认的版本就会被编译器创建。这个默认版本简单地把每个成员从目标对象复制到当前对象，当然这肯定不是你所需要的，因此你需要重写它来提供你自己的行为，或者重载它让这个赋值操作符在当前类型之外的类型上也能工作。

就示例8-15中的Balance类来说，你或许与下面一样来定义这个赋值操作符：

```
Balance& operator=(const Balance& other) {
   val_ = other.val_;
   return(*this);
}
```

如果你不熟悉操作符重载的话，第一件使你困惑的事情就是这个operator=语法。这是声明所有操作符的方法；你可以把每一个operator都想象成一个名为operator[*symbol*]的函数，这里这个symbol是你需要重载的操作符。操作符和普通函数之间唯一不同的是它的调用语法。事实上，如果你愿意写一些额外的代码并且使得代码难看的话，你可以使用这个语法来调用操作符，如下所示：

```
x.operator=(y);  // Same thing as x = y, but uglier
```

我的赋值操作符的操作实现得很简单。它使用另一个参数上的值来更新当前对象的val_成员，并且返回指向当前对象的索引。赋值操作符作为索引返回当前对象，因此调用者就可以在表达式中使用这个赋值操作符：

www.TopSage.com

大家网 TopSage.com

```
Balance x, y, z;
// ...
x = (y = z);
```

这样，这个从(y = z)中返回的值就是修改后的对象y，然后这个y对象被赋值操作符传给对象x。这在赋值操作符中没有与它在算术运算中那样常见，但是为了遵循这个传统，你还是应该返回一个指向当前对象的索引（我讨论这个问题是由于它与算术运算相关）。

尽管简单的赋值只是开始；最有可能的是你想使用其他的算术操作符来定义一些更有趣的行为。表8-1中列出了所有的算术和赋值操作符。

表8-1 算术和赋值操作符

操作符	行为
=	赋值（必须是成员函数）
+	加
+=	
-	减
-=	
*	乘或者是取指针指向的值
*=	
/	除
/=	
%	模运算
%=	
++	自增
--	自减
^	按位异或
^=	
~	按位取余
&	按位与
&=	
\|	按位或
\|=	
<<	左移位
<<=	
>>	右移位
>>=	

www.TopSage.com

对于表8-1中的大部分操作符来说，都有两个特征：第一个是它可以按照传统的方式那样来使用的版本，例如，1+2，第二个是可以把一个操作符结果的值赋给一个变量，例如，x+=5。请注意在8.13节中说明了自增++和自减--操作符。

每一个算术或者是赋值运算符的实现都非常相同，除了赋值操作符有点例外之外，因为它不能使一个独立的函数（例如，它必须是一个成员函数）。

加法操作符对于重载来说是非常流行的一种选择，特别是由于它能用在数学以外的一些上下文中，如把一个字符串追加到另一个字符串中，因此让我们首先来考虑一下加法操作符。它把右边参数加到左边参数上然后把结果赋给左边的参数，如下这些代码：

```
int i = 0;
i += 5;
```

当第二行代码执行完后，整数i的值就被加上了5。相同地，如果你查看示例8-15，你也期望下面的这些代码具有相同的行为：

```
Balance checking(500.00), savings(23.91);
checking += 50;
```

这就是说，你期望当这个+=操作符被执行之后，这个checking的值增加了50。使用示例8-15中的实现，这些都是精确地发生的。请看下面给+=操作符定义的函数：

```
Balance& operator+=(double other) {
   val_ += other;
   return(*this);
}
```

对于赋值操作符，它的参数列表就是作为它的右值提供给它的；在这个例子中，是一个整型。这个函数体是很小的：这里所做的就是把这个参数值加到这个私有成员变量上。当所有的工作完成后，返回*this指针。你应该从赋值和算术成员操作符中返回*this指针，这样它们就可以作为表达式使用，这个表达式的值可以作为别的方面的输入。例如，如果我这样来声明operator+=：

```
void operator+=(double other) { // Don't do this
   val_ += other;
}
```

然后某个人可能会在某个地方这样来在我的类上使用这个操作符并且把它的结果传给别的函数：

```
Balance moneyMarket(1000.00);
// ...
updateGeneralLedger(moneyMarket += deposit); // Won't compile
```

这就会产生一个问题，因为Balance::operator+=返回void无类型，并且像updateGeneralLedger这样的函数期望得到一个Balance对象。如果你从算术和赋值成员操作符返回当前对象，那么你就不会有问题。尽管这并不适用于所有的操作符。其他的操作符（如数组下标操作符[]或者是关系操作符&&）就是返回一个对象而不是*this指针，因此这就是为算术和赋值成员操作符应保存的方针。

www.TopSage.com

这些赋值操作符应关心的也是算术操作符应该关心的，但算术操作符关心的也是赋值操作符应该注意的吗？另一个使用算术操作符的方法如下所示：

```
int i = 0, j = 2;
i = j + 5;
```

在这种情况下，j被加上5并且结果还被赋给i（这样，如果i是一个对象并且i还不是一个本地类型的话，就需要使用到i的类型中的赋值操作符），但j则是没有被改变的。如果你需要在你的类中也有同样的行为，你就可以把加法操作符重载成一个独立的函数。例如，你可能需要如下代码从而使得它更有意义：

```
Balance checking(500.00), savings(100.00), total(0);
total = checking + savings;
```

你可以通过两步来这样做。第一步就是创建一个函数来重载+操作符：

```
Balance operator+(const Balance& lhs, const Balance& rhs) {
   Balance tmp(lhs.val_ + rhs.val_);
   return(tmp);
}
```

这个函数带两个常量型Balance对象，给它的两个私有成员求和，再创建一个临时对象然后返回它。请注意，不像赋值操作符，它返回一个对象而不是一个对象的索引。这是因为这个返回的对象是临时的，并且如果返回一个引用的话就意味着调用者将拥有一个指向一个不再存在的变量的引用。尽管这样这个函数将不能由于它自己而工作，因为它需要访问它的参数的私有成员（假设你使得数据成员变成非公有）。为了达到这个目的，Balance类就不得不作为一个友元来声明这个函数：

```
class Balance {
   // These have to see private data
   friend Balance operator+(const Balance& lhs, const Balance& rhs);
   // ...
```

任何作为友元声明的东西都可以访问一个类中的所有成员，因此这就像是玩了一个魔术一样。仅仅要记住的是要把它的参数声明成常量型的，因为你不想对象被修改。

这虽然能解决大部分问题，但是还是不能解决所有的问题。你的类的用户可能如下这样把表达式放在一起：

```
total = savings + 500.00;
```

这将与示例8-15中的代码一起工作，因为编译器可以看到Balance类有一个带浮点数的构造函数，因此就使用这个构造函数为这个500.00浮点数创建一个临时的Balance对象。尽管这有两个问题：创建一个临时对象的代价，以及Balance并没有可用于可以被加法操作符使用的所有可能参数的构造函数。让我们来假设你有一个名为Transaction的类，它代表贷方和借方数目。Balance类的用户可能如下这样来使用：

```
Transaction tx(-20.00);
total = savings + tx;
```

www.TopSage.com

这将不能编译通过，因为没有操作符来把一个Balance对象加到一个Transaction对象上。因此就需要创建一个：

```
Balance operator+(const Balance& lhs, const Transaction& rhs) {
   Balance tmp(lhs.val_ + Transaction.amount_);
   return(tmp);
}
```

尽管这需要一些额外的工作。你不得不把这个操作符声明成这个Transaction的友元，并且为了使得你能够以任何的参数顺序来使用这个+操作符或者是使得这个操作符是可以交换的，你还需要以相反顺序的参数来创建一个完全相同的版本，

```
Balance operator+(const Transaction& lhs, const Balance& rhs) {
   Balance tmp(lhs.amount_ + rhs.val_);
   return(tmp);
}
```

通过同样的标记，当一个构造函数被自动调用时，如果你想避免临时对象的产生，你可以创建你自己的操作符来处理任何类型的变量：

```
Balance operator+(double lhs, const Balance& rhs) {
   Balance tmp(lhs + rhs.val_);
   return(tmp);
}
Balance operator+(const Balance& lhs, double rhs) {
   Balance tmp(lhs.val_ + rhs);
   return(tmp);
}
```

再一次，你又需要创建两个这样的函数来让如下的表达式能够工作正常：

```
total = 500.00 + checking;
```

在这种情况下，这个临时对象的构造是很小的，并且相对来说，代价也不大。但是一个临时对象仍然还是一个临时对象，并且在一个简单的单一语句中，它不会产生可观的代价，但是在一个宽广的环境中，你还是应该考虑这微不足道的优化 —— 如果上百万个这种临时对象创建出来会如何呢？因为用户可能想给一个vector<Balance>矢量中的每一个元素都增加的话就有可能产生上百万个这种临时对象。所以最好的办法就是知道这个类通常是如何被使用的并且如果你不确定的话，就测试一下它在性能上的代价。

在这一点上，产生这样的疑问是合理的：为什么需要为这些非赋值算术操作符创建独立函数，而不是如赋值操作符那样使用成员函数呢？事实上，在你感兴趣的类上你可以把这些类型的操作符声明为成员函数，但是这样就会使得这些操作符不具有可交换性。为了使得一个操作符在用户自定义类型上具有可交换性，你将不得不在这些可能在这个操作符中涉及的类上把它声明为成员函数，并且这将可以工作（虽然这样每一个这种类都知道每一个其他类的内部成员），但是它在那些你可以使用内在类型的操作符上还是不能工作，除非存在这样的构造函数可以使用，并且在这种情况下，你还不得不为临时对象付出代价。

www.TopSage.com

C++中，操作符重载是一个强大的功能，并且与多重继承一样，它有支持者也有反对者。事实上，大部分其他的流行语言根本就不支持操作符重载。然而如果你小心地使用它，它还是能使得你的类非常强大和精炼。

大部分的标准操作符都有一些传统的意义，并且总的来说，你还是应该遵循它们的传统上的意义。例如，这个<<操作符是左移位，或者如果你正处理流的话，它的意思就是写入，如下所示：

```
cout << "This is written to the standard output stream.\n";
```

如果你决定给你的一个或多个类都重写这个操作符<<，你应该使得它能够处理这两件事情中的一种，或者最少类似于它们的事情。重载一个操作符是一件事，但给予一个操作符一个完全新的语义就是另外一件事。除非你正给你的应用程序或者类库中通篇引入一个新的惯例（这其实仍然不是一个好主意），并且这个操作符仍然会给除了你之外的其他人它的直觉意义，你还是应该坚持它的标准意义。

为了有效地重载操作符，有大量的前期工作需要做。但是你仅需要这样做一次，以后每一次在一个简单的表达式中使用你的类时都能得到好处。如果你适当地和明智地使用操作符重载，它就能使你的代码变得容易读和写。

请参考8.13节。

8.15 调用一个超类虚函数

问题

如何调用一个特定类的超类中的函数？这个函数在子类中被重写了，因此通常的p->method()将不能达到你需要的效果。

解决方案

使用你的目标基类来修饰你的成员函数名；例如，你有两个类，如示例8-16所示：

示例8-16 调用虚函数的特定的版本

```
#include <iostream>

using namespace std;

class Base {
public:
   virtual void foo() {cout << "Base::foo()" << endl;}
};

class Derived : public Base {
public:
   virtual void foo() {cout << "Derived::foo()" << endl;}
};
```

www.TopSage.com

大家网 TopSage.com

```
int main() {
   Derived* p = new Derived();

   p->foo();  // Calls the derived version
   p->Base::foo(); // Calls the base version
}
```

讨论

使用C++中正常的多态技术来实现这个需求不是一个好的主意，但是有很多时候你还是需要这样做。和C++中许多别的技术相比，当你想调用一个虚函数的特定基类版本时，它的语法是比较复杂的，需要使用你需要的那个类的名字来修饰，在示例8-16中如下：

```
p->Base::foo();
```

这就能调用定义在Base类中的foo函数，而不是指针p所指向的Base类的子类中的那个foo函数。

第9章

异常与安全性

9.0 概述

本章包含了如何使用C++的异常处理机制的几个技术。C++在异常处理上有很强的支持，并且通过使用这些技术，你可以写出有效处理异常并且很容易调试的代码。

9.1节描述了C++中抛出和捕获异常的语意，并且解释如何来写一个代表异常的类。如果你没有或只有很少的处理异常经验的话，这是一个非常好的起点。它也描述了一些定义在<stdexcept>和<exception>中的标准异常类。

其他的几节将要说明一些如何最佳地使用异常的技术，并且它们定义了几个要用的关键术语。当一些不期望的事情发生时，仅仅只是抛出一个异常，或者捕获一个异常来仅仅是打印一个错误信息并且终止程序都不是一个好的软件。为了有效地使用C++的异常处理机制，当一个异常抛出时，你不得不写一些不泄漏任何资源的代码，并且或者是当一个异常抛出时，运行一些事先定义好的行为。这就是基本的和强效的异常安全保证机制。我说明的这些你能使用的技术使你的构造函数和各种成员函数得到了保证。

9.1 创建一个异常类

问题

如何为异常的抛出和捕获创建你自己的异常类？

解决方案

在一些简单的需求中，你可以实践一下抛出和捕获C++的任何类型，也即是它有合法的复制构造函数和析构函数。异常的处理是一个复杂的主题，因此当设计一个代表异常的环境时需要考虑多方面的事情。示例9-1说明了一个简单的异常类。

www.TopSage.com

示例9-1 一个简单的异常类

```
#include <iostream>
#include <string>

using namespace std;

class Exception {

public:
   Exception(const string& msg) : msg_(msg) {}
  ~Exception() {}

   string getMessage() const {return(msg_);}
private:
   string msg_;
};

void f() {
   throw(Exception("Mr. Sulu"));
}

int main() {

   try {
 f();
   }
   catch(Exception& e) {
      cout << "You threw an exception: " << e.getMessage() << endl;
   }
}
```

讨论

C++中支持异常的三个关键字是：try、catch和throw。它的语法如下：

```
try {
   // Something that may call "throw", e.g.
   throw(Exception("Uh-oh"));
}
catch(Exception& e) {
   // Do something useful with e
}
```

C++（Java和C#等语言）中的异常是一个在程序的某个点放置一个信息的方法，并且希望某个人能沿着某处的调用栈来检查你的信息。它是一些简单技术，如返回一个错误代码或信息的一个替代技术。异常使用的语意（如“trying”某些代码，“throwing”一个异常并且“catching”异常）和C++的别的一些操作是明显不同的，因此在描述如何创建一个异常类之前，我还要说明一下什么是异常和抛出或捕获一个异常又意味着什么。

当一个异常情况发生时，你想调用的代码应该能够意识到它，并通过throw语句抛出一个消息，如下所示：

```
throw(Exception("Something went wrong"));
```

当你这样做时，这个运行时环境就会构造一个异常对象，然后它开始展开调用栈直到它发现一个已经进入但还没退出的try代码段。如果运行时环境不能发现一个这样的代码段，也就意味着需要抛到main函数（或者当前线程的最高级范围）并不能继续展开调用

www.TopSage.com

栈的话，那么一个名为terminate的函数就会被调用。但是如果这样的一个try代码段被发现了的话，它就查看这个try代码段中的每个catch语句以查找与抛出的异常类型相同的catch语句。catch语句如下所示：

```
catch(Exception& e) { //...
```

在这一点上，异常类的复制构造函数从这个被抛出的异常上创建了一个新的异常（在throw抛出的那个作用域中的那个异常是一个临时的异常对象，因此编译器可能会把它优化掉）。由于超出了它的作用域，这个原始的异常就会被销毁掉，并且执行catch块中的代码。

在这个catch块代码中，如果你想继续传播这个你刚刚捕获到的异常，你可以不带参数地调用throw：

```
throw;
```

这将继续沿着调用栈进行异常的处理过程直到发现另一匹配的处理。这就允许每个地方都捕获这个异常并且对它做一些有用的处理，然后处理完（也可以还没有处理完）时重新把这个异常抛出。

这就是异常的抛出和捕获的基本技术。既然现在你已经了解这些基本的知识，就考虑一下示例9-1。你可以使用一个字符指针或一个字符串来构造一个异常，然后把这个异常抛出。但是这样的一个简单的类基本没什么用，因为它只是起到了封装一个文本信息的作用。事实上，在这种情况下，你通过使用一个字符串而不是一个异常对象就能达到同样的效果：

```
try {
   throw(string("Something went wrong!"));
}
catch (string& s) {
   cout << "The exception was: " << s << endl;
}
```

这并不是一个很好的办法；但我这样做的目的只是为了证明一下异常的本质：异常可以是任何的C++类型。只要你愿意，你就可以抛出一个整型、字符、类、结构或任意其他的C++类型。但是你最后是使用异常类的继承或者那些标准类库，或者是你自己的异常层次关系。

使用一个异常类的层次关系的最大的好处就是它能让你通过异常类本身来表达这个异常环境的本质，而不是一个错误代码、文本字符串、严重程度或者别的什么。这就是标准类库通过定义在<stdexcept>的标准异常所能做的。这些定义在<stdexcept>的异常类的基类事实上是定义在<exception>中的异常。图9-1中说明标准异常类中的类层次关系。

www.TopSage.com

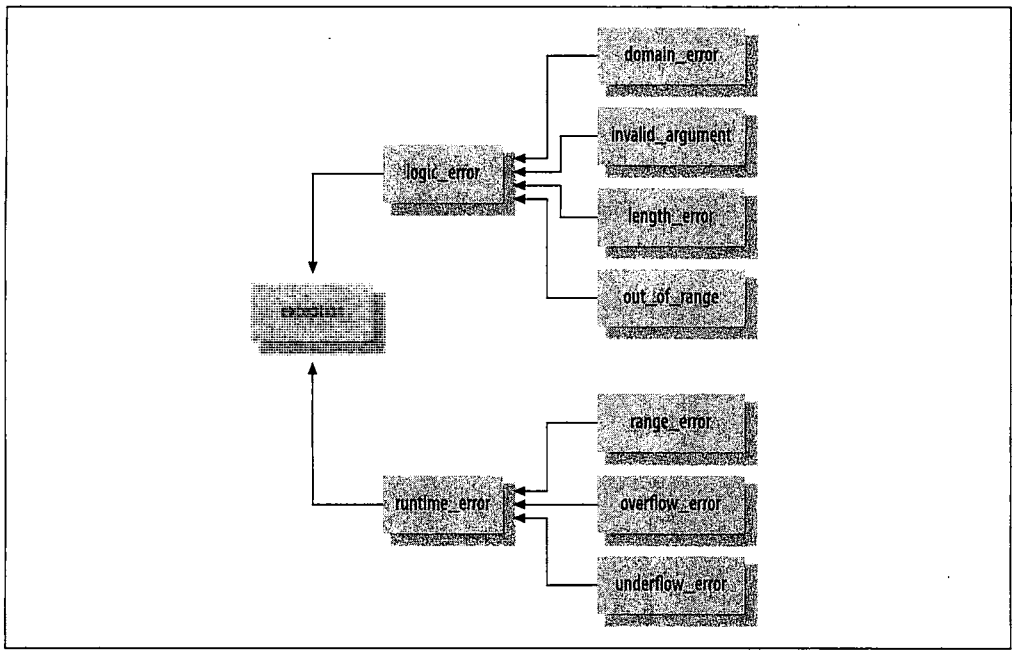


图 9-1 标准异常类层次关系

每个标准异常类通过它的名字就能标识它所代表的那一类情形。例如，类logic_error所代表的是在写代码或代码审查时就应该捕获的一些情况，并且它的子类就是代表这些情况的子类型：如违反了先验条件，提供了一个超出范围的索引下标，提供了一个不合法的参数等。这些逻辑错误的一个补充是一些运行时错误，它由runtime_error来标识。它所标识的一些情况是，或更有可能的是那些不能在编码阶段捕获到的一些错误，如超出了规定的范围、溢出或下溢等。

这只是异常情形的一个有限的集合，并且标准的异常类可能不能满足你的全部要求。你有可能还需要一些跟某个应用相关的异常，如database_error、network_error、painting_error等。稍后我将讨论。但是之前让我们来讨论一下标准异常是如何工作的。

由于标准类库使用了这些标准的异常类，所以在有问题的情况下，你可以使用这些标准类库中的类来抛出一个异常，例如，如果试着去索引那些超过了矢量末端的下标对应的对象的话：

```
std::vector<int> v;
int i = -1;

// fill up v...

try {
   i = v.at(v.size()); // One past the end
```

www.TopSage.com

TopSage.com

```
}
catch (std::out_of_range& e) {
   std::cerr << "Whoa, exception thrown: " << e.what() << '\n';
}
```

如果你输入了一个小于0或大于siz()-1的下标给矢量的at函数，那么vector<>::at就会抛出一个out_of_range的异常。既然你已经知道了这个情况，你就可以写一个处理器来处理这种情形下的异常。如果你不希望处理某个特殊的异常，但是你想使用相同的方式来处理所有的异常，你可以捕获所有异常的基类：

```
catch(std::exception& e) {
   std::cerr << "Nonspecific exception: " << e.what(  ) << '\n';
}
```

这样做的话，就能捕获所有从exception继承的所有类。它提供了一个虚函数可以提供实现时才定义的信息字符串。

我饶了一个大圈子。跟在示例9-1后的这些讨论都是为了证明异常的好处。在两方面可以使得一个异常类有用：通过类的层次关系，类能传达异常的本质并且通过捕获异常能给用户显示一个信息。异常类的层次关系允许那些使用类库的开发人员来实现安全的、容易调试的代码，并且通过这个异常的信息文本能够让开发人员给应用程序的终端用户显示一些有意义的错误信息。

总的来说，异常是一个复杂的主题，并且安全地、有效地处理异常是软件工程中最困难的工作的一部分，并且C++又更为特别。你如何来实现一个构造函数，在它的函数体中有异常的抛出或初始化列表中有异常的抛出等情况下同时又不会带来内存泄漏？一个异常安全又意味着什么？在接下来的章节中，我将就这些问题展开讨论。

9.2 创建一个异常安全的构造函数

问题

如何创建一个异常安全的构造函数？

解决方案

在构造过程中，如果一个异常抛出，在构造函数中使用try和catch机制来正确地清除相应的资源。示例9-2说明了一个简单的例子Device和Broker类。Broker在堆上构造了两个Device对象，但如果在这个构造的过程中有一个异常抛出的话，则需要能够正确地把这两个Device对象清除掉。

示例9-2 一个异常安全的构造函数

```
#include <iostream>
#include <stdexcept>

using namespace std;
```

www.TopSage.com

```
class Device {
public:
   Device(int devno) {
      if (devno == 2)
         throw runtime_error("Big problem");
   }
  ~Device() {}
};

class Broker {

public:
   Broker (int devno1, int devno2) :
      dev1_(NULL), dev2_(NULL) {
      try {
         dev1_ = new Device(devno1);  // Enclose the creation of heap
         dev2_ = new Device(devno2);  // objects in a try block...
      }
     catch (...) {
        delete dev1_; // ...clean up and rethrow if
        throw;   // something goes wrong.
      }
   }
  ~Broker() {
      delete dev1_;
      delete dev2_;
   }

private:
   Broker();
   Device* dev1_;
   Device* dev2_;
};

int main() {

   try {
 Broker b(1, 2);
   }
   catch(exception& e) {
 cerr << "Exception: " << e.what() << endl;
   }
}
```

讨论

构造函数、成员函数、析构函数或其他的函数都是异常安全的，这就保证了不会有任何资源会泄漏，并且也不会使得它的对象处在不一致的状态中。C++ 中，这两类保证分别叫做基本和健壮的保证。

基本的异常安全保证是非常直接的。也就是说当一个异常被抛出时，当前的操作将不会泄漏任何资源并且这个操作涉及到的对象将仍然可以使用（也即是你可以调用其他的成员函数并把它销毁掉，例如，它不会处在一个被销毁的状态）。也就是程序将还是处在一个一致的状态下，尽管它可能不是你期望的状态。这个规则是直接的：如果在一个成员函数体的任何地方都可以抛出一个异常，堆对象不会被泄漏并且这个操作涉及的任何对象都可以被销毁或被调用者重新设置。另外一种保证叫做健壮的异常安全保证，使得如果一个操作失败的话，对象的状态将保持不变。这种保证应用在一个对象的后置构造操作中，因为，在定义上，在构造过程中就抛出一个异常的话，这个对象是不可能完全构

www.TopSage.com

造出来的，也就不可能处在一个有效的状态中。在9.4节中我将回到成员函数这个主题上。但是现在让我们专注于构造上。

示例9-2中定义了两个类：Device和Broker，虽然它们的功能有限，但是它们可以代表任何的设备/代理这种应用类型，在这种应用中，你有一些类用来打开两个设备中的每一种的连接并且管理它们之间的通信。在只有一个可用设备的情况下，代理是没有用的，因此当你实例化一个代理时，你需要事务性的语义，也就是如果这两个设备中的任何一个被访问时有异常抛出的话，另一个也要被释放。这就能保证内存或其他资源都没有被泄漏。

try和catch能够处理这项工作。在这个构造函数中，在一个try代码块中包含堆对象的分配并且在catch代码块中捕获在构造过程中抛出的任何东西，如下所示：

```
try {
   dev1_ = new Device(devno1);
   dev2_ = new Device(devno2);
}
catch (...) {
   delete dev1_;
   throw;
}
```

这个catch子句中的省略号代表抛出的任何东西都会被捕获。这就是你需要的，因为你正在做的就是在有什么出错的情况下能自动清除，然后不管是什么类型异常被抛出来都接着再把它抛出去。你需要再把这个异常抛出，因为这样正在实例化这个代理对象的客户代码就能够给这个异常进行一些有用的处理，如在一些地方输出一些错误信息。

在这个catch代码块中，我只删除了dev1_，因为只有在给dev2_创建一个对象时才可能抛出一个异常。如果抛出一个异常，dev2_将不会赋值，因此我不需要删除它。然而，如果你在初始化dev2_后还做了一些处理的话，你还是需要保证能够清除它的。例如：

```
try {
   dev1_ = new Device(devno1);
   dev2_ = new Device(devno2);
   foo_ = new MyClass();  // Might throw
}
catch (...) {
   delete dev1_;
   delete dev2_;
   throw;
}
```

在这个例子中，你不用担心删除一个没有赋任何值的指针是否有问题（只要你首先正确地初始化了它们），因为删除一个NULL指针没有任何作用。换句话说，如果在给dev1_赋值时抛出了一个异常，你的捕获处理代码仍会调用来删除dev2_，但只要在这个初始化列表中你已经把它初始化为NULL了就没有问题。

就像我在9.1节中说的那样，设计一个可靠、灵活的异常处理策略是有难度的，但异常安

www.TopSage.com

全保证是没有区别的。更详细的关于如何设计异常安全的代码的技术，请参考由 Herb Sutter(Addison Wesley)写的《C++ 异常》这本书。

请参考 9.3 节。

9.3 创建一个异常安全的初始化列表

问题

如何创建一个异常安全的初始化列表？

解决方案

使用 try 和 catch 的一个特殊的语法来捕获从初始化列表中抛出的异常。如示例 9-3 中所示。

示例 9-3 在初始化列表中处理异常

```
#include <iostream>
#include <stdexcept>

using namespace std;

// Some device
class Device {
public:
   Device(int devno) {
      if (devno == 2)
         throw runtime_error("Big problem");
   }
  ~Device() {}
private:
   Device();
};

class Broker {

public:
   Broker (int devno1, int devno2)
      try : dev1_(Device(devno1)),   // Create these in the initializer
            dev2_(Device(devno2)) {} // list.
      catch (...) {
         throw; // Log the message or translate the error here (see
                // the discussion)
      }
  ~Broker() {}

private:
   Broker();
   Device dev1_;
   Device dev2_;
};

int main() {

   try {
      Broker b(1, 2);
   }
   catch(exception& e) {
```

www.TopSage.com

```
      cerr << "Exception: " << e.what() << endl;
   }
}
```

讨论

这个在初始化列表中处理异常的语法看上去和传统的C++语法有些不同，因为它把try代码块当成构造函数体。示例9-3中的主要部分是Broker的构造函数：

```
Broker (int devno1, int devno2)   // Constructor header is the same
   try :                          // Same idea as a try {...} block
         dev1_(Device(devno1)),   // The initializers follow
         dev2_(Device(devno2)) {
       // This is the constructor body.
   }
   catch (...) {                  // The catch handler is *after*
 throw;                           // the constructor body
   }
```

try和catch的行为和你期望的相同；它与通常的try代码块唯一的不同就是当你想捕获一个从初始化列表中抛出的异常时，try处于初始化列表中冒号的前面，而冒号后面仍然是初始化列表，并且这个try子句也是构造函数的一部分。如果在初始化列表或构造函数体中有任何东西抛出，构造函数体后面的catch处理块就会把它捕获住。在构造函数体中，如果有必要的话，你仍然可以嵌入try/catch代码块，但这种嵌套的用法常常让人觉得怪异。

此外，在把成员变量的初始化移到初始化列表中时，因为另外的原因示例9-3和示例9-2还是有些不同的。这个Device对象成员这一次并不是使用new操作符在堆上创建的。我这样做是为了说明关于安全和成员对象的几点事实。

首先，使用栈对象而不是使用堆对象能让编译器提供它的内置安全性。在构造过程中，如果在初始化列表中的任意对象都抛出异常的话，这个对象的内存能够自动地释放，因为在异常的处理过程中栈是会自动展开的。第二，并且是更好的，任何在构造过程中被成功创建的对象都可以在不需要你亲自捕获这个异常并显式地删除它的情况下自动释放。

但或许你需要或者更喜欢堆对象。考虑在示例9-2中原来的那个Broker类。你可以仅仅在初始化列表中初始化你的指针，对吗？

```
class BrokerBad {

public:
   BrokerBad (int devno1, int devno2)
      try : dev1_(new Device(devno1)),   // Create heap objects with
            dev2_(new Device(devno2)) {} // initializers
      catch (...) {
         if (dev1_) {
            delete dev1_; // Shouldn't compile, and
            delete dev2_; // is a bad approach if it
         }                // does
         throw; // Rethrow the same exception
      }
```

www.TopSage.com

大家网 TopSage.com

```
  ~BrokerBad(  ) {
      delete dev1_;
      delete dev2_;
  }

private:
   BrokerBad(  );
   Device* dev1_;
   Device* dev2_;
};
```

这里存在两个问题。第一，不会成功地编译，因为构造函数的catch块不会让你的程序代码访问你这个成员变量，因为在这个点上，它们还不存在。第二，即使你的编译器允许，它也是一个不好的用法。考虑这种情况，当构造dev1_的对象时构造函数抛出了一个异常。下面这段在catch处理代码块中的代码将会被执行：

```
catch (...) {
   if (dev1_) {     // What value does this contain?
      delete dev1_; // Now you are deleting an undefined value
      delete dev2_;
   }
   throw;           // Rethrow the same exception
}
```

如果在构造这个dev1_的过程抛出了一个异常，而new操作符又还没有返回这个新分配内存的地址的话，那么dev1_的值就还没有被改变。那么它包含什么呢？它是没定义的，因为它还没有被某个值初始化。这样，当你调用delete dev1_时，你就可能删除了一个垃圾指针地址，那么你的程序就会崩溃，由此你将被开除，并且在你以后的生活中，你就觉得颜面扫地。

为了避免这种改变生活的惨败，你需要自初始化列表中把你的指针初始化为NULL，并且在你的构造函数中创建一个堆对象。这样的话，就很容易捕获那些出问题的东西并清理掉它们，因为删除一个NULL指针是没问题的。

```
BrokerBetter (int devno1, int devno2) :
dev1_(NULL), dev2_(NULL) {
   try {
      dev1_ = new Device(devno1);
      dev2_ = new Device(devno2);
   }
   catch (...) {
     delete dev1_; // This will always be valid
     throw;
   }
}
```

因此，作为总结，如果你必须使用指针成员，在初始化列表中你应把它初始化为NULL，然后在构造函数中使用try/catch块给它分配对象。你可以在catch块中取消分配任何的内存。然而，如果你使用自动类型的成员的话，就要在初始化列表中构造它们并需要使用特殊的try/catch来处理任意的异常。

请参考9.2节。

www.TopSage.com

9.4 创建异常安全的成员函数

问题

如何实现一个成员函数，并且你需要这个成员函数支持基本的和增强的异常安全保证？也就是说它不会泄漏任何资源，并且当一个异常抛出时，它不会让某个对象处在一个不合法的状态中。

解决方案

意识到什么操作可以抛出异常然后首先处理它，常常是使用try/catch块。一旦这个可以抛出异常的代码被执行，你就更新对象的状态。示例9-4给出了一个使得成员函数异常安全的例子。

示例9-4 一个异常安全的成员函数

```
class Message {

public:
   Message(int bufSize = DEFAULT_BUF_SIZE) :
      bufSize_(bufSize),
      initBufSize_(bufSize),
      msgSize_(0),
      buf_(NULL) {
      buf_ = new char[bufSize];
   }

  ~Message() {
      delete[] buf_;
   }

   // Append character data
   void appendData(int len, const char* data) {
      if (msgSize_+len > MAX_SIZE) {
         throw out_of_range("Data size exceeds maximum size.");
      }
      if (msgSize_+len > bufSize_) {

         int newBufSize = bufSize_;
         while ((newBufSize *= 2) < msgSize_+len);

         char* p = new char[newBufSize];     // Allocate memory
                                             // for new buffer

         copy(buf_, buf_+msgSize_, p);       // Copy old data
         copy(data, data+len, p+msgSize_);   // Copy new data

         msgSize_ += len;
         bufSize_ = newBufSize;

         delete[] buf_;  // Get rid of old buffer and point to new
         buf_ = p;
      }
      else {
         copy(data, data+len, buf_+msgSize_);
         msgSize_ += len;
      }
   }
```

www.TopSage.com

```
   // Copy the data out to the caller's buffer
   int getData(int maxLen, char* data) {
      if (maxLen < msgSize_) {
         throw out_of_range("This data is too big for your buffer.");
      }
      copy(buf_, buf_+msgSize_, data);
      return(msgSize_);
   }

private:
   Message(const Message& orig) {} // We will come to these
   Message& operator=(const Message& rhs) {} // in Recipe 9.5
   int bufSize_;
   int initBufSize_;
   int msgSize_;
   char* buf_;
};
```

讨论

示例9-4中的Message类是一个支持字符数据的类；你可以使用它来给文本自动换行或者当成二进制数据来从一个系统传到另一个系统。这里感兴趣的是它的成员函数appendData，它把调用者的数据追加到已在缓冲区中的数据中，如果有必要的话就增大缓冲区。它支持增强的异常安全保证，尽管当你第一眼看到它时觉得不明显。

请看appendData函数部分：

```
if (msgSize_+len > bufSize_) {

   int newBufSize = bufSize_;
   while ((newBufSize *= 2) < msgSize_+len);

   char* p = new char[newBufSize];
```

这段代码就是用来增大缓存的。我以原来缓存大小的两倍增长它直到它足够大。这段代码是异常安全的，因为这段代码唯一可以抛出异常的就是new操作符，并且在new操作符之前我没有更新对象的状态或者分配任何其他的资源。如果操作系统不能够分配这个所需大小的内存的话，它将抛出bad_alloc异常。

如果内存分配成功，然后我通过复制数据并更新成员变量才开始更新对象的状态：

```
copy(buf_, buf_+msgSize_, p);
copy(data, data+len, p+msgSize_);

msgSize_ += len;
bufSize_ = newBufSize;

delete[] buf_;
buf_ = p;
```

这些操作没有一个是会抛出异常的，这里我们很清楚（这仅仅是因为在缓存中的数据是一个字符的序列；更多的解释请看示例9-5中的讨论）。

这个解决方案是简单的，并且它是使得成员函数具有增强的异常安全的保证的一个总的策略：首先处理任何可以首先抛出异常的部分，然后当所有这些危险的事情处理完后，就

www.TopSage.com

可以稍微放松地来更新对象的状态了。appendData成员函数使用了一个临时变量来存储缓存的大小。这解决了存储缓存大小的问题，但它真的支持基本异常安全保证而不会泄漏任何资源吗？是的，但很有限。

复制给序列中每个正在被复制的元素调用了操作符=。示例9-4中，每个元素都是一个字符，因此它是安全的，因为每一个字符赋给另一个简单的赋值操作是不可能抛出任何异常的。但是我说它很有限，因为你不应该由于这个特殊的异常安全个案就使得你想象在赋值超过了范围时也不会有异常抛出。

想象片刻，使用一个比较小的字符缓存，你不得不写一个包含任意值的一个数组的Message类。你可能想如下一样来把它实现成一个类模板。

示例9-5 类的一个泛型版本

```
template<typename T>
class MessageGeneric {

public:
   MessageGeneric(int bufSize = DEFAULT_BUF_SIZE) :
      bufSize_(bufSize),
      initBufSize_(bufSize),
      msgSize_(0),
      buf_(new T[bufSize]) {}

  ~MessageGeneric() {
      delete[] buf_;
   }

   void appendData(int len, const T* data) {
     if (msgSize_+len > MAX_SIZE) {
        throw out_of_range("Data size exceeds maximum size.");
     }
     if (msgSize_+len > bufSize_) {

        int newBufSize = bufSize_;
        while ((newBufSize *= 2) < msgSize_+len);

        T* p = new T[newBufSize];

        copy(buf_, buf_+msgSize_, p);// Can these throw?
        copy(data, data+len, p+msgSize_);

        msgSize_ += len;
        bufSize_ = newBufSize;

        delete[] buf_;  // Get rid of old buffer and point to new
        buf_ = p;
     }
     else {
        copy(data, data+len, buf_+msgSize_);
        msgSize_ += len;
     }
   }

   // Copy the data out to the caller's buffer
   int getData(int maxLen, T* data) {
      if (maxLen < msgSize_) {
         throw out_of_range("This data is too big for your buffer.");
      }
      copy(buf_, buf_+msgSize_, data);
```

www.TopSage.com

```
      return(msgSize_);
   }

private:
   MessageGeneric(const MessageGeneric& orig) {}
   MessageGeneric& operator=(const MessageGeneric& rhs) {}
   int bufSize_;
   int initBufSize_;
   int msgSize_;
   T* buf_;
};
```

现在你不得不更加小心，因为你对于目标类型不能做任何假设。例如，你如何知道T::operator=不会抛出异常呢？你不可能，因此你不得不对这些可能性有所准备。

在try代码块中调用copy函数：

```
try {
   copy(buf_, buf_+msgSize_, p);
   copy(data, data+len, p+msgSize_);
}
catch(...) {    // I don't care what was thrown; all I know
   delete[] p; // is that I have to clean up after myself,
   throw; // then rethrow.
}
```

由于你使用省略号来捕获任何可能抛出的异常，因此你就能保证如果T::operator=抛出异常的话，也能捕获住，并且你可以清理分配的堆内存。

严格地说，copy函数事实上不会抛出任何东西，但T::operator=操作会抛出。这是因为copy（以及其他一些标准类库中的算法）总的来说是异常中立的，也就是说无论它调用什么抛出的异常，它都会把它传到它的调用者那里，而不会自己把它捕获住并不再重新抛出。它保持了这个权利：捕获异常，做一些清理工作，然后再重新抛出这个异常，但是最后的这个通过标准类库中的类或函数抛出的异常都会在它的调用者那里被处理。

使得一个成员函数是异常安全是一项繁琐的工作。它需要你考虑所有的可能抛出异常的地方并且需要正确的方法来处理。什么时候能够抛出异常呢？函数的调用能发生在任何地方。对一些本地数据类型的操作不能抛出异常，并且析构函数不应该抛出异常，但是其他的（可能是一个独立的函数、成员函数、操作符、构造函数等等）都可能是异常的源发地。示例9-5和示例9-6都是处理窄范围比较的异常的例子。这个类包含了很少的成员变量，并且这些类的行为也是比较离散的。随着成员函数和成员变量数目的增加，并且随着你又引进继承和虚函数等，要保持增强的异常安全保证就会变得更具有挑战性。

最后，基于你大部分应用程序的需求，你仅需要保证你所需的那种异常安全程度。换句话说，如果你正在写一个产生web页面的对话程序，你的开发计划可能就包含必要的研发和测试来使得它有比较强的异常安全保证就可以。并且，对于你的客户来说，碰到几个偶尔的、不明确的错误信息（如“Unknown error, aborting”等）还是可以接受的。但另一方面，如果你正在写一个控制直升机转子的角度的程序，你的客户将会需要更加安全的保证，对这几个偶尔的不明确的错误信息就不能接受。

www.TopSage.com

大家网 TopSage.com

9.5 安全地复制一个对象

问题

如何实现基本的类复制操作：复制构造函数和赋值操作都是异常安全的？

解决方案

使用9.4节中的技术，首先处理那些可能抛出异常的操作，然后等那些危险的操作完成后使用那些不会抛出异常的操作来改变对象的状态。示例9-6中重新定义了Message类，并定义了赋值操作符和复制构造函数。

示例9-6 异常安全的赋值和复制构造函数

```
#include <iostream>
#include <string>

const static int DEFAULT_BUF_SIZE = 3;
const static int MAX_SIZE     = 4096;

class Message {

public:
   Message(int bufSize = DEFAULT_BUF_SIZE) :
      bufSize_(bufSize),
      initBufSize_(bufSize),
      msgSize_(0),
      key_("") {
      buf_ = new char[bufSize]; // Note: now this is in the body
   }

  ~Message() {
      delete[] buf_;
   }

   // Exception-safe copy ctor
   Message(const Message& orig) :
      bufSize_(orig.bufSize_),
      initBufSize_(orig.initBufSize_),
      msgSize_(orig.msgSize_),
      key_(orig.key_) { // This can throw...

      buf_ = new char[orig.bufSize_];  // ...so can this
      copy(orig.buf_, orig.buf_+msgSize_, buf_); // This can't
   }

   // Exception-safe assignment, using the copy ctor
   Message& operator=(const Message& rhs) {

      Message tmp(rhs);    // Copy construct a temporary
      swapInternals(tmp); // Swap members with it
      return(*this);       // When we leave, tmp is destroyed, taking
                           // the original data with it
   }

   const char* data() {
      return(buf_);
   }

private:
   void swapInternals(Message& msg) {
      // Since key_ is not a built-in data type it can throw,
```

www.TopSage.com

```
        // so do it first.
        swap(key_, msg.key_);

        // If it hasn't thrown, then do all the primitives
        swap(bufSize_,msg.bufSize_);
        swap(initBufSize_, msg.initBufSize_);
        swap(msgSize_,msg.msgSize_);
        swap(buf_,     msg.buf_);
    }
    int bufSize_;
    int initBufSize_;
    int msgSize_;
    char* buf_;
    string key_;
};
```

讨论

这个复制构造函数和这个私有的成员变量swapInternals处理了所有的工作。这个复制构造函数在初始化列表中初始化所有的私有成员和一个非私有成员。然后分配一个新的缓存并把数据复制到这个缓存中。非常的简单，但使用什么样的顺序来初始化呢？你可能会说这个初始化的顺序不就是按这个初始化列表中先后顺序的，但这样的话你就打开了通向错误的大门。

例如，你或许想把缓存的分配也放到初始化列表中，如下所示：

```
Message(const Message& orig) :
    bufSize_(orig.bufSize_),
    initBufSize_(orig.initBufSize_),
    msgSize_(orig.msgSize_),
    key_(orig.key_),
    buf_(new char[orig.bufSize_]) {
    copy(orig.buf_, orig.buf_+msgSize_, buf_);
}
```

你也许期望一切正常，因为如果在这个初始列表中缓存的分配失败的话，所有其他已经构建好的对象就会被销毁掉。但是这种行为是不能有保证的，因为初始化列表中成员初始化的顺序是按它们在类的头文件中声明的顺序，而不是它们在初始化列表中出现的顺序。这个成员的声明顺序如下所示：

```
int bufSize_;
int initBufSize_;
int msgSize_;
char* buf_;
string key_;
```

结果，buf_将在key_前被初始化。如果在key_的初始化时抛出了异常的话，buf_就不会被销毁掉，并且你就将创建一大块没有引用的内存。你可以通过在构造函数中使用try/catch块来保证这点（如9.2节），但是把buf_的初始化放到构造函数体中更简单，因为构造函数体的执行是发生在初始化列表之后的。

调用copy函数是不会抛出异常的，因为它正在复制原始值。但这就是异常安全保证的微妙之处：如果它复制的是一个对象的话，就有可能抛出异常（如，它是一个T元素的泛型容器），这样你就能捕获这个异常并删除相关的内存。

www.TopSage.com

你想复制对象的另一个办法就是使用赋值操作符operator=。由于它和复制构造函数具有同样的需求（如，使我的成员等于我的参数成员），重用已有的代码使你的生活容易一点。唯一另外的就是通过私有成员来交换数据以使得事情简单一点。我期望我发明了这种技术，但是我还是不得不把我的信任交给Herb Sutter和Stephen Dewhurst，因为我首先看到了他们的著作。

第一眼看到也许你就明白了，但是我要解释的并不仅仅是这个案例。考虑第一行，它通过复制构造函数来构造一个临时的对象tmp：

```
Message tmp(rhs);
```

现在我们创建了rhs的一个克隆对象。tmp对象等于rhs。现在，你就可以使它和this指针所指的当前对象交换数据。

```
swapInternals(tmp);
```

过一会我们再来讨论Stephen Dewhurst函数。现在，我们关心的是this对象的成员等于tmp对象的成员。并且tmp就是rhs的一个副本，因此this对象就等于rhs对象。但是等等：我们将一直具有这个临时对象。这没问题，当你返回*this时，tmp就被自动销毁了，因为超过了它的作用域，同时旧的对象就被返回了。

```
return(*this);
```

就是这样。但是这异常安全吗？构造tmp对象时是异常安全的，因为我们的构造函数是异常安全的。swapInternals函数完成了大部分工作，因此让我们来讨论一下它是否是异常安全的。

swapInternals交换当前对象中的数据成员和传入它的对象的数据成员。它使用swap来达到这个功能，swap带两个参数*a*和*b*，并且创建一个临时的*a*的副本，然后把*b*赋给*a*，再把这个临时*a*的副本赋给*b*。这样，它是异常安全的并且是异常中立的，因为唯一出现的异常就是来自这些被操作的对象。它没有动态的内存操作，因此它支持基本的异常保证，即不会泄漏任何资源。

由于key_不是原始的数据类型，也就是说在它身上的操作有可能抛出异常，我首先交换它。这样，如果有异常抛出，别的其他对象就不会被销毁。当然这样并不能保证key_对象本身不会被销毁。当你在对象上执行操作时，你自己才是异常安全的保证。如果没有异常抛出的话，我就可以在家自由自在，因为我知道原始数据的交换是不会抛出异常的。因此，swapInternals函数能够满足异常安全的基本和增强的要求。

这就产生一个有趣的问题。如果你有多个对象成员的话，会如何？如果你有两个字符串成员，swapInternals函数的开始如下所示：

```
void swapInternals(Message& msg) {
   swap(key_, msg.key_);
   swap(myObj_, msg.myObj_);
   // ...
```

www.TopSage.com

这就有个问题：如果这第二个交换抛出一个异常的话，我们该如何来保证安全地回滚第一个交换呢？

换句话说，现在key_被新值成功地更新了，但myObj_的交换失败了，因此key_的值就是脏数据。如果调用者捕获了这个异常并且处理的话，尽管什么都没发生，这里和前面还是有一些不同。把key_复制到一个临时字符串first是一种方法，但是这不能保证就是安全的，因为这样的复制可能抛出一个异常。

一种绕开这个问题的办法就是使用堆对象：

```
void swapInternals(Message& msg) {
   // key_ is a string* and myObj_ is a MyClass*
   swap(key_, msg.key_);
   swap(myObj_, msg.myObj_);
```

当然，这就意味着你有更多的动态内存需要管理，但是使得异常安全的保证常常影响你的设计，因此，在设计过程的早期就考虑异常的处理是一个好的主意。

这一节的主题和前一节一样都是关于异常安全的。首先处理可能产生问题的工作，等着try/catch处理完以免有异常的事情发生，并且如果发生了异常的话，就自动完成清理工作。如果一切正常，你就可以悠闲地更新对象的状态了。

请参考9.2节和9.3节。

www.TopSage.com

第10章

数据流与文件

10.0 概述

数据流是C++标准库中最有威力且最复杂的组件之一。为那种原始的没有任何格式的输入输出来使用它们总的来说是直接的，但是使用这种标准的操作来改变格式以适合你的需求或者实现你自己的操作则不是那么直接。因此，这一章的前面几节将描述格式化输出流的方法。其余的几节将描述如何实现和一个流的类的对象或者如何从一个流对象中读数据。

然后这些章节将介绍在文件或目录上的一些读和写操作。如果你的程序需要使用文件，特别是如果你的程序是一个守护进程或服务器端的进程的话，你就很有可能需要创建文件和目录，清除文件和目录等等。这些章节将解释如果使用C++来完成这些单调而又必须处理的任务。

这些章节的第三部分将描述如何使用标准的字符串成员函数来操作文件和路径本身。标准的字符串包含很多的检查和操作字符串本身的成员函数，并且如果你需要分析路径和文件名的话，这些成员函数也是很方便的。如果你的需求在这些章节中没有涉及到的话，请参考一下第7章，也许你能发现在那一章中就讨论了你的需求。

文件的操作需要和操作系统直接的交互，并且不同的操作系统之间是存在差别的（并且有时还很不兼容）。大部分典型的文件和目录操作需求的是标准C系统调用的一部分，并且在不同的系统上基本相同。当在不同的操作系统上这些功能存在差别时，我将指出。

就像我在前面章节中讨论的那样，Boost类库是一个源项目，并且它是一个高质量的可移植的类库。但是由于本书的主题是有关C++而不是Boost类库的，所以在有可能的情况下，我都使用标准的C++解决方案。然而在很多情况下（大部分都在10.12节中指出）都不存在标准的C++解决方案，因此我就使用由Beman Dawes实现的Boost文件系统类库，

www.TopSage.com

大家网 TopSage.com

这个类库提供了一个可移植的文件系统接口，从而提供了一个可移植的解决方案。如果你需要实现一个可移植的文件系统交互，请参考一下Boost文件系统类库，它能让你节省一些时间和精力。更多的关于Boost项目的信息，请参考*www.boost.org*。

10.1 文本输出对齐

问题

如何垂直排列你的文本输出？例如，如果你想导出表中的数据，你可能希望输出的格式如下所示：

```
Jim          Willcox     Mesa            AZ
Bill         Johnson     San Mateo       CA
Robert       Robertson   Fort Collins    CO
```

你或许想这些输出数据是右对齐或左对齐的。

解决方案

使用在<ostream>中为窄字符或宽字符定义的ostream或者wostream的输出流，并且使用标准的流操作符来设置字段的宽度并对齐文本。示例10-1给出了一个简单的例子。

示例10-1 对齐文本输出

```
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

int main() {

   ios_base::fmtflags flags = cout.flags();
   string first, last, citystate;
   int width = 20;

   first = "Richard";
   last  = "Stevens";
   citystate = "Tucson, AZ";

   cout << left            // Left-justify in each field
   << setw(width) << first // Then, repeatedly set the width
   << setw(width) << last  // and write some data
   << setw(width) << citystate << endl;

   cout.flags(flags);
}
```

输出如下：

```
    Richard   Stevens   Tucson, AZ
```

讨论

一个操作符就是操作在流上的函数。通过操作符<<将这些操作函数应用在流上。一个流

www.TopSage.com

(输入或输出)的格式是由最基本的流类ios_base上的一序列标志和设置来控制的。操作函数的存在就是为调整这些标志和设置提供一个便利的方法，这样用户就不用显式通过setf或者标志本身来设置，而这样的方法对于写和读来说都是麻烦、不方便的。格式化输出的最好的办法就是使用这些操作函数。

示例10-1使用了两个操作函数来把文本排列输出到列中。这个setw操作函数就是设置字段的宽度，并且让这个字段中的数据左对齐(与左对齐相对的就是右对齐)。一个字段的意义就是能让你的输出在某个边被填补或者保证你的数据值是这个字段中唯一输出的数据。就像在示例10-1中，你左对齐数据值然后再设置字段的宽度，接下来你向这个流中写入的值就从这个字段的第一个字符开始。如果你送到这个流中的数据的宽度没有这个字段的设置宽度那么宽，那么这个字段的右边就会使用这个流的填充字符来填充，默认的情况下，这个填充字符是一个简单空格。你也可以使用setfill操作函数来改变这个默认的填充字符，如下所示：

```
myostr << setfill('.') << "foo";
```

如果你送到字段中的值比字段的宽度更宽，那么就没有填充字符被加上并且你的整个值都会输出。

表10-1中包含那些操作在任意类型(文本、浮点型、整型等)值上的操作函数的总结。并且同时还有一序列的只能应用在浮点型上的操作函数，在10.2节中描述它们。

表10-1 文本操作函数

操作函数	描述	输出例子
left right	在当前字段的宽度内左对齐或右对齐数据值，并且使用填充字符来填充余下的空格	左对齐： apple bananna cherry 右对齐(使用一个10字符宽的字段)： apple bananna cherry
setw(int n)	设置n个字符的字段的宽度	请看前面的例子
setfill(int c)	使用字符c来填充余下的空格	cout << setfill('.') << setw(10) << right << "foo" 输出如下： foo

www.TopSage.com

表 10-1 文本操作函数（续）

操作函数	描述	输出例子
boolalpha noboolalpha	显示布尔值 true 和 false 代替 1 和 0 来代表当前的本地环境	cout << boolalpha << true 输出如下： true
endl	输出一个新行到流中并且刷新输出缓冲区	n/a
ends	输出一个 null 字符（'\0'）到流中	n/a
flush	刷新缓冲区	n/a

表 10-1（和下一节的表 10-2）中的一些操作函数绑定了一些二进制流标志，并且事实上是实现了两个操作形式：流标志打开和关闭。例如 boolalpha，如果你希望布尔值的显示是当前环境中的值（如，“true” 和 “false”），就使用 boolalpha 操作函数。如果你关闭这种功能，就使用 noboolalpha，那么 0 和 1 就代替布尔值输出，这种行为是默认的。

所有的这些操作函数都会一直生效直到它被显式改变，但 setw 例外。在示例 10-1 中，你可以看到每一次的输出我都调用这个 setw，而 left 则只被调用一次。这是因为当每一个值使用操作符 << 输出到流以后，宽度都会被重新设置成零；为了保证每一个字段的输出都使用同样的宽度，我不得不每次都调用 setw。

标准的操作提供了很多函数，但是它们并不能处理任何事情。如果你想实现你自己的操作，请参考 10.2 节。

就像标准类库中其他类型字符的类那样，这些操作函数也能在使用窄字符或宽字符的流上工作。因此，你可以用一个模板来使用它们从而实现那些能操作在任意类型字符类型上的格式化函数。示例 10-2 中实现了一个类模板 TableFormatter，它把数据格式化到一个相等宽度的列中并输出到流中。

示例 10-2 用于表格式数据的泛型类

```
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>

using namespace std;

// TableFormatter formats data for output to a stream of characters
// of type T.
template<typename T>
class TableFormatter {

public:
   TableFormatter(basic_ostream<T>& os) : out_(os) {}
  ~TableFormatter() {out_ << flush;}
```

www.TopSage.com

```
    template<typename valT>
    void writeTableRow(const vector<valT>& v, int width);
    //...

private:
  basic_ostream<T>& out_;
};

template<typename T>     // refers to class template param list
template<typename valT> // refers to mem fn template param list
void TableFormatter<T>::writeTableRow(const std::vector<valT>& v,
                                      int width) {

    ios_base::fmtflags flags = out_.flags();

    out_.flush(  );
    out_ << setprecision(2) << fixed;  // Set the precision, in case
                                       // this is floating-point data
    for (vector<valT>::const_iterator p = v.begin();
         p != v.end(  ); ++p)
      out_ << setw(width) << left << *p;  // Set the width, justify,
                                          // and write the element
    out_ << endl;// Flush
    out_.setf(flags); // Set the flags back to normal
}

int main() {

    TableFormatter<char> fmt(cout);

    vector<string> vs;

    vs.push_back( "Sunday" );
    vs.push_back( "Monday" );
    vs.push_back( "Tuesday" );

    fmt.writeTableRow(vs, 12);
    fmt.writeTableRow(vs, 12);
    fmt.writeTableRow(vs, 12);

    vector<double> vd;

    vd.push_back(4.0);
    vd.push_back(3.0);
    vd.push_back(2.0);
    vd.push_back(1.0);

    fmt.writeTableRow(vd, 5);
}
```

示例 10-2 的输出如下：

```
    Sunday Monday Tuesday
    4.00 3.00 2.00 1.00
```

请参考表 10-1 和 10.2 节。

10.2 格式化浮点数输出

问题

为了精度原因（相对于固定小数点）或者为了更容易读简单地垂直排列的十进制小数点，如何使用一个定义好的格式来输出浮点数？

www.TopSage.com

解决方案

使用<iomanip>和<ios>提供的标准操作来控制写入流中的浮点数值的格式。这里会涉及到各种不同的方法，示例 10-3 中提供了几个不同的方法来显示 pi 的值。

示例 10-3 格式化 pi

```
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

int main() {

   ios_base::fmtflags flags =  // Save old flags
      cout.flags();

   double pi = 3.14159265;

   cout << "pi = " << setprecision(5)  // Normal (default) mode; only
        << pi << '\n';                 // show 5 digits, including both
                                       // sides of decimal point.

   cout << "pi = " << fixed       // Fixed-point mode;
        << showpos                // show a "+" for positive nums,
        << setprecision(3)        // show 3 digits to the *right*
        << pi << '\n';            // of the decimal.

   cout << "pi = " << scientific  // Scientific mode;
        << noshowpos              // don't show plus sign anymore
        << pi * 1000 << '\n';

   cout.flags(flags);  // Set the flags to the way they were
}
```

它的输出如下：

```
pi = 3.1416
pi = +3.142
pi = 3.142e+003
```

讨论

特定的操作浮点数输出的操作符可分为两类。一类是这一章中讨论的设置格式操作符，总体来说是设置浮点数和整型值的显示格式，另一类是每一个中显示格式的微调。格式如下：

Normal (the default)

在这种格式中，显示的数字位数是固定的（默认是6位），并且小数点右边的数字位数是变化的。因此，默认的情况下，pi 将显示为 3.14159，而 pi 和 100 的乘积就显示为 314.159。

Fixed

在这种格式下，小数点右边显示的数字位数是固定的，而小数点左边显示的位数则不是固定的。在这种情况下，使用默认的6位精度，pi 将显示为 3.141593，而 pi

www.TopSage.com

和100的乘积就显示为314.159265。在这两种情况下，显示在小数点右边的数字位数都是6位，而显示的所有数字位数可以无限地增长。

Scientific

这种显示格式就是一个单一的数字后面跟着小数点，而小数点后面再跟着由精度设定的那么多个位数的数字，然后紧跟着就是字母“e”和这个数的10的幂。在这种情况下，pi和1000的乘积将显示为3.141593e+003。

表10-2说明了所有的影响浮点数输出的操作符（总的来说是数字输出）。那些能够和这些浮点数操作符一起使用的一般的操作符请参考表10-1。

表10-2 浮点和数字操作符

操作函数	描述	示例输出
Fixed	使用小数点右边的固定位数输出浮点数。	默认的精度是6位。 pi = 3.141593
scientific	使用科学计数法来输出浮点数，也即是小数点左边只有一个十进制数而小数点右边则是设置的精度位数和乘数。	pi * 1000，默认的精度是6位小数位。 pi = 3.141593e+003
setprecision	控制显示的位数（请参考后面的解释）。	Pi在默认的情况下显示三位数。 pi = 3.14 固定格式 pi = 3.142 科学计数法的方式： pi = 3.142e+000
showpos noshowpos	在正数前显示+。任意类型的数、十进制或整数	+3.14
showpoint noshowpoint	显示小数点，即使小数点后只有一个0。这种方式只能是浮点数，不能是整数	接下来这行以两位的精度显示： cout << showpoint << 2.0 输出为： 2.00
showbase noshowbase	显示基数：十进制、八进制（0开头）、十六进制（0x开头）。参考下面的函数	十进制：32 八进制：040 十六进制：0x20
dec oct hex	为十进制、八进制或十六进制数设置基数。默认情况下，这个基数是不显示的；使用showbase来显示这个基数	参考前面的例子
Uppercase nouppercase	以大写的方式显示值	它们也设置数字的输出格式，如0X用于十六进制数，E用于科学计数方式

www.TopSage.com

在这所有的三种输出形式中，除了setprecision之外的所有操作符都具有相同的效果。在默认的模式中，精度指的是小数点两边的数字位。例如，使用默认精度为2来显示pi，如下所示：

```
cout << "pi = " << setprecision(2)
     << pi << '\n';
```

输出如下：

```
pi = 3.1
```

相反地使用固定小数点格式显示pi：

```
cout << "pi = " << fixed
     << setprecision(2)
     << pi << '\n';
```

输出如下：

```
pi = 3.14
```

这是因为在固定小数点格式中，精度指的是小数点右边的数字位数。如果你使用这种模式来输出pi与1000的乘积，小数点右边的数字位数将会没有任何变化：

```
cout << "pi = " << fixed
     << setprecision(2)
     << pi * 1000 << '\n';
```

输出：

```
pi = 3141.59
```

这很好，因为你可以设置你的精度，使用setw来设置你的字段的宽度，使用right来右对齐你的输出（请参考10.1节），并且你的十进制小数点还能全部垂直地对齐。

由于一个操作符是在流上设置格式标志的一个方便的方法，但请记住这些设置标志将一直有效直到你取消它们或者把这个流销毁掉。请记住在开始给某个流设置标志时保存以前设置的标志，而当你完成你的工作时则恢复原来的标志。

请参考10.3节。

10.3 编写自己的流操作程序

问题

如何实现一个流操作符来完成那些标准的流不能完成的工作，或者你需要单个操作符来在一个流上设置多个标志而不是每次都是调用多个操作符？

解决方案

为了实现一个不带参数的操作符，写一个带一个ios_base作为参数的函数并且给这个

www.TopSage.com

参数设上流标志。如果你需要一个带参数的流，请参考稍后的讨论。示例10-4说明了如何实现一个不带参数的流操作符。

示例10-4 一个简单的流操作符

```
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

// make floating-point output look normal
inline ios_base& floatnormal(ios_base& io) {
   io.setf(0, ios_base::floatfield);
   return(io);
}
int main() {

   ios_base::fmtflags flags =  // Save old flags
      cout.flags();

   double pi = 3.14159265;

   cout << "pi = " << scientific   // Scientific mode
        << pi * 1000 << '\n';

   cout << "pi = " << floatnormal
        << pi << '\n';

   cout.flags(flags);
}
```

讨论

这里有两类流操作符：带参数的和不带参数的。不带参数的流容易实现。你需要做的就是写一个函数，这个函数带一个流作为参数，然后再在这个流上做一些处理（设置一些标志或者改变一些设置），然后再把这个流返回。而实现一个带一个或多个参数的流操作符则更为复杂，因为你需要再实现附加的类或函数，这些附加的类或函数需要操作在后面的这些场景上。由于不带参数的流操作符实现起来简单，就让我们以它作为讨论的开始。

当你读完10.1节时，你或许已经意识到了有三种浮点型格式并且只有两个操作符来选择这些格式。这个默认的格式没有对应的操作符；你不得不在流上设置某个标志来返回默认的格式，如下所示：

```
myiostr.setf(0, ios_base::floatfield);
```

但是为了一致性和方便，你可能想增加一个你自己的流操作符来完成相同的工作。这就是示例10-4所做的。这个`floatnormal`操作符设置一个合适的流标志并以默认的格式输出浮点型数据。

编译器知道使用你的新函数来做什么，因为标准类库已经给`basic_ostream`流定义了操作（`basic_ostream`是这个类模板的名字，`ostream`和`wostream`都是这个类模板的实例。如下所示：

www.TopSage.com

```
basic_ostream<charT,traits>& operator<<
(basic_ostream<charT,traits>& (* pf)(basic_ostream<charT,traits>&))
```

这个例子中，pf是一个函数指针，它带一个basic_ostream 引用作为参数并且返回一个basic_ostream引用。这个操作只是在这个参数指定的流上调用你的函数。

实现一个带参数的流操作则更为复杂。那是为什么呢？请考虑一下不带参数的流操作符是如何工作的。当你像下面一样使用一个流操作符时：

```
myostream << myManip << "foo";
```

你不带括号地使用它，因此事实上你解析的是你的操作符函数的地址。操作符<<事实上是调用你的操作符函数，并且传到流中去，因此你的操作符才能完成它的工作。

为了比较的缘故，假设你有一个带一个数字参数的流操作符，因此，理想的情况下，你可以如下使用它：

```
myostream << myFancyManip(17) << "apple";
```

它是如何工作的呢？如果你认为myFancyManip是一个带一个整型参数的函数的话，那么就会有问题：你如何不在参数中包含一个流并且不显式使用它而把这个流传给这个函数呢？下面也许是你想做的：

```
myostream << myFancyManip(17, myostream) << "apple";
```

但这是笨拙和冗余的。一个流操作符的方便之处就在于它能在一行代码中使用多个操作符<<，并且读和使用都很容易。

这个解决方案就是要找到一个绕开这个问题的方案。代替在这个流上用操作符<<调用你的操作符，你需要引入一个临时的能返回这个操作符 << 可以使用的东西。

首先，你需要定义一个临时的类来完成这项工作。为了简单起见，假设你需要实现一个名为setWidth的操作符，它完成和setw相同的工作。你需要实现的这个临时类的结构如下所示：

```
class WidthSetter {

public:
   WidthSetter (int n) : width_(n) {}
   void operator()(ostream& os) const {os.width(width_);}
private:
   int width_;
};
```

这个类的功能是简单的。使用一个整型参数来构造它，并且当通过一个流参数来调用括号操作符时，就在这个流上设置它的宽度为初始化时设定的值。这种方案的要点在于WidthSetter将要通过一个参数来构造并被另一个使用。你的操作符函数将就是这个构造它的函数，并且应该如下一样来使用：

```
WidthSetter setWidth(int n) {
   return(WidthSetter(n));    // Return the initialized object
}
```

www.TopSage.com

它所做的就是返回一个用这个整型数初始化的WidthSetter对象。这个操作符就是你可以与操作符<<一起使用的操作符，如下所示：

```
myostream << setWidth(20) << "banana";
```

但是仅是这样还不够，因为如果setWidth只是返回一个WidthSetter对象的话，操作符<<并不知道如何做。你还需要重载操作符<<，这样它才知道该如何处理一个WidthSetter：

```
ostream& operator<<(ostream& os, const WidthSetter& ws) {
   ws(os);// Pass the stream to the ws object
   return(os); // to do the real work
}
```

这种办法解决了这个问题，但不是一个通用的方案。你不必为了每一个你实现的可接受参数的操作符都写一个这种与WidthSetter相类似的类（也许你愿意，但不是一个明智的办法），因此一个更好的办法就是使用模板和函数指针来构建一个好的通用的基础结构，并且任意别的操作符都可以以它作为基础。示例10-5给出了一个ManipInfra类和一个操作符<<的版本，它们使用模板参数来处理一个流可以处理的各种类型的字符和一个流操作符可以使用的任意种类的参数。

示例10-5 操作符基础结构

```
#include <iostream>
#include <string>

using namespace std;

// ManipInfra is a small, intermediary class that serves as a utility
// for custom manipulators with arguments. Call its constructor with a
// function pointer and a value from your main manipulator function.
// The function pointer should be a helper function that does the
// actual work. See examples below.
template<typename T, typename C>
class ManipInfra {

public:
   ManipInfra (basic_ostream<C>& (*pFun)
               (basic_ostream<C>&, T), T val)
     : manipFun_(pFun), val_(val) {}
   void operator()(basic_ostream<C>& os) const
      {manipFun_(os, val_);}  // Invoke the function pointer with the
private:                       // stream and value
   T val_;
   basic_ostream<C>& (*manipFun_)
      (basic_ostream<C>&, T);
};

template<typename T, typename C>
basic_ostream<C>& operator<<(basic_ostream<C>& os,
                             const ManipInfra<T, C>& manip) {
   manip(os);
   return(os);
}

// Helper function that is ultimately called by the ManipInfra class
ostream& setTheWidth(ostream& os, int n) {
   os.width(n);
```

www.TopSage.com

大家网 TopSage.com

```
   return(os);
}

// Manipulator function itself.  This is what is used by client
// code
ManipInfra<int, char> setWidth(int n) {
   return(ManipInfra<int, char>(setTheWidth, n));
}

// Another helper that takes a char argument
ostream& setTheFillChar(ostream& os, char c) {
   os.fill(c);
   return(os);
}

ManipInfra<char, char> setFill(char c) {
   return(ManipInfra<char, char>(setTheFillChar, c));
}

int main() {

  cout << setFill('-')
  << setWidth(10) << right << "Proust\n";
}
```

如果这个事件序列你还是不清晰的话，我建议你在调试器中运行一下示例10-5。一旦你看到了这些动态的动作，你就会明白。

10.4 使类可写入流

问题

为了给人读或者是为了永久保存，例如序列化，如何把一个类写到一个输出流中？

解决方案

重载操作符 << 把合适的数据成员写入到流中。示例10-6说明了是如何这样做的。

示例10-6 把对象写到一个流中

```
#include <iostream>
#include <string>

using namespace std;

class Employer {
   friend ostream& operator<<                // This has to be a friend
     (ostream& out, const Employer& empr); // so it can access non-
public:                                      // public members
   Employer() {}
  ~Employer() {}

   void setName(const string& name) {name_ = name;}
private:
   string name_;
};

class Employee {
   friend ostream& operator<<
     (ostream& out, const Employee& obj);
public:
```

www.TopSage.com

```
    Employee() : empr_(NULL) {}
   ~Employee() {if (empr_) delete empr_;}

    void setFirstName(const string& name) {firstName_ = name;}
    void setLastName(const string& name) {lastName_ = name;}
    void setEmployer(Employer& empr) {empr_ = &empr;}
    const Employer* getEmployer() const {return(empr_);}

private:
    string firstName_;
    string lastName_;
    Employer* empr_;
};

// Allow us to send Employer objects to an ostream...
ostream& operator<<(ostream& out, const Employer& empr) {

    out << empr.name_ << endl;

    return(out);
}

// Allow us to send Employee objects to an ostream...
ostream& operator<<(ostream& out, const Employee& emp) {

    out << emp.firstName_ << endl;
    out << emp.lastName_ << endl;
    if (emp.empr_)
     out << *emp.empr_ << endl;

    return(out);
}

int main() {

    Employee emp;
    string first = "William";
    string last = "Shatner";
    Employer empr;
    string name = "Enterprise";
    empr.setName(name);

    emp.setFirstName(first);
    emp.setLastName(last);
    emp.setEmployer(empr);

    cout << emp;  // Write to the stream
}
```

讨论

你需要做的第一件事就是把操作符<<声明为你需要输出到流中的类的友元。你应该使用操作符<<来代替一个成员函数，如writeToStream(ostream& os)，因为左移操作符是把标准类库中的任何别的东西写入流的惯例。你需要把它声明为友元，是因为在大部分情况下，你需要把私有的成员写到流中，但是非友元函数是不能访问它们的。

然后，你就需要定义你的操作符<<是操作在一个ostream上还是wostream上（定义在<ostream>中），并且定义你声明这个操作符为友元的类。在这个类的定义中，你需要决定哪些数据成员需要输出到流中去。典型地，你可能需要把所有的数据成员输出到流中，就像我在示例10-6中做的那样：

www.TopSage.com

```
out << emp.firstName_ << endl;
out << emp.lastName_ << endl;
```

示例 10-6 中，我通过调用操作符 << 来把指针 empr_ 所指的对象输出：

```
if (emp.empr_)
   out << *emp.empr_ << endl;
```

我可以这样做是因为empr_指向了Employer类的一个对象，就像Employee类那样，我也给这个 Employer 类定义了 << 操作符。

当你正把你的类成员输出到流中时，你的<<操作符必须返回这个传入的流。无论你什么时候重载这个 << 操作符，你都应该这样做，这样后面的操作才能进行下去，如下例：

```
cout << "Here's my object: " << myObj << '\n';
```

我给出的这个方法是简单的，如果你仅是为了消遣的话把一个类输出到一个输出流中，这个例子是能够正常工作的，但是它仅仅完成了一小部分工作。如果你需要把一个对象写到一个流中，常常有两个原因。或者是这个流在某个地方需要被人来读（cout、控制窗口、日志文件等），或者是这个流是一个临时或永久的存储介质（如stringstream、网络连接、文件等），并且将来你打算从这个流中重新组织对象。如果你需要从一个流中重新产生对象（10.5 节的主题）的话，你就需要仔细考虑一下你的类之间的关系。

给任何东西实现序列化都是一个困难的工作。如果一个类引用（通过指针或引用）了别的类，就像任何平凡的类那样，当你输出任何一个对象时，你都需要考虑潜在的循环引用，然后当把它们读进来时你还需要能正确地构造它们。如果你不得不从头开始来构建它们的话，你还必须对它们的设计做一些考虑，但是如果你可以使用第三方的类库的话，你应该试着使用 Boost 序列化类库，它以一种可移植的方式为对象序列化提供了一个框架。

请参考 10.5 节。

10.5 使流可读取类

问题

如何向一个流中写入了某个类的一个对象，然后从这个流中把这些数据读进来，并且使用这些数据来初始化这个类的一个对象？

解决方案

使用操作符>>来从这个流中读入这些数据到你的类中从而组装它的数据成员，这和示例 10-6 中的实现刚好相反。请看示例 10-7 中的实现。

www.TopSage.com

示例 10-7 从一个流中读入一个对象

```
#include <iostream>
#include <istream>
#include <fstream>
#include <string>

using namespace std;

class Employee {
   friend ostream& operator<<     // These have to be friends
      (ostream& out, const Employee& emp); // so they can access
   friend istream& operator>>     // nonpublic members
      (istream& in, Employee& emp);

public:
   Employee() {}
  ~Employee() {}

   void setFirstName(const string& name) {firstName_ = name;}
   void setLastName(const string& name) {lastName_ = name;}

private:
   string firstName_;
   string lastName_;
};

// Send an Employee object to an ostream...
ostream& operator<<(ostream& out, const Employee& emp) {

   out << emp.firstName_ << endl;
   out << emp.lastName_ << endl;

   return(out);
}

// Read an Employee object from a stream
istream& operator>>(istream& in, Employee& emp) {

   in >> emp.firstName_;
   in >> emp.lastName_;

   return(in);
}

int main() {

   Employee emp;
   string first = "William";
   string last = "Shatner";

   emp.setFirstName(first);
   emp.setLastName(last);

   ofstream out("tmp\\emp.txt");

   if (!out) {
      cerr << "Unable to open output file.\n";
      exit(EXIT_FAILURE);
   }

   out << emp;  // Write the Emp to the file
   out.close();

   ifstream in("tmp\\emp.txt");

   if (!in) {
      cerr << "Unable to open input file.\n";
```

www.TopSage.com

```
        exit(EXIT_FAILURE);
    }

    Employee emp2;

    in >> emp2;  // Read the file into an empty object
    in.close();

    cout << emp2;
}
```

讨论

使得一个类可以从一个流中读进来的步骤几乎和把一个对象写入流的步骤相同，但是功能上是相反的。如果你还没有读10.4节的话，你应该先读一下，这样就能更好地理解示例10-7。

首先，你需要给你的目标类声明一个友元操作符>>，但是在这种情况下你需要使用istream而不是ostream。然后定义这个操作符>>（而不是操作符<<）来直接把数据从流中读到你的类成员变量中。当你正读数据时，返回这个输入流。

请参考10.4节。

10.6 获取文件信息

问题

如何获取文件的一些信息，如文件的大小、对应的设备、最后的修改时间等？

解决方案

使用定义在<sys/stat.h>中的C系统调用stat。示例10-8给出了这个系统调用的典型应用，它打印文件的一些属性。

示例10-8 获取文件的信息

```
#include <iostream>
#include <ctime>
#include <sys/types.h>
#include <sys/stat.h>
#include <cerrno>
#include <cstring>

int main(int argc, char** argv )
{
    struct stat fileInfo;

    if (argc < 2) {
        std::cout << "Usage: fileinfo <file name>\n";
        return(EXIT_FAILURE);
    }

    if (stat(argv[1], &fileInfo) != 0) {  // Use stat( ) to get the info
        std::cerr << "Error: " << strerror(errno) << '\n';
        return(EXIT_FAILURE);
    }
```

www.TopSage.com

大家网 TopSage.com

```
    std::cout << "Type:     : ";
    if ((fileInfo.st_mode & S_IFMT) == S_IFDIR) { // From sys/types.h
       std::cout << "Directory\n";
    } else {
       std::cout << "File\n";
    }

    std::cout << "Size: " <<
       fileInfo.st_size << '\n';// Size in bytes
    std::cout << "Device    : " <<
       (char)(fileInfo.st_dev + 'A') << '\n';  // Device number
    std::cout << "Created   : " <<
       std::ctime(&fileInfo.st_ctime);         // Creation time
    std::cout << "Modified  : " <<
       std::ctime(&fileInfo.st_mtime);         // Last mod time
}
```

讨论

C++标准库使用流来支持对文件内容的操作，但是它没有内在的支持读或改变操作系统管理的文件的原数据，如文件的大小、所有者、权限、各种时间戳等信息。但是标准C则包含很多标准的系统调用，你可以通过这些标准C的系统调用来获取文件的这些信息，如示例10-8中使用的那样。

获取文件信息有两部分。第一，有一个名为stat的结构，它包含很多用来存储文件信息的成员，第二个就是与这个结构stat同名的系统调用，通过这个系统调用你可以获取你需要的有关文件的信息，并且通过这个系统调用你还可以填充这个结构。一个系统调用就是一个提供来自操作系统服务的函数。标准C中包含各种各样的系统调用，并且它们中的大部分都是各种不同Unix版本的标准。这个stat结构如下（来自Kernigan和Richie编著的《C编程语言》）：

```
struct stat {
   dev_t    st_dev;     /* device of inode */
   ino_t    st_ino;     /* inode number */
   short    st_mode;    /* mode bits */
   short    st_nlink;   /* number of links to file */
   short    st_uid;     /* owner's user id */
   short    st_gid;     /* owner's group id */
   dev_t    st_rdev;    /* for special files */
   off_t    st_size;    /* file size in characters */
   time_t   st_atime;   /* time last accessed */
   time_t   st_mtime;   /* time last modified */
   time_t   st_ctime;   /* time inode last changed */
};
```

这个stat结构的每一个成员的含义依赖于操作系统。例如，st_uid和st_gid在Windows系统上就没有任何意义；但在Unix系统中，它们事实上包含这个文件属主的用户id和用户组id。有关哪个值被支持并且它又是什么含义你需要查看你的操作系统的文档。

示例10-8说明如何显示这个stat结构中的一些可移植的成员。st_mode包含描述文件类型的位掩码，你可以用它来识别一个文件是目录与否。st_size则用位来表示文件的大小。第三个成员time_t则是这个文件的访问、修改和创建时间戳。

www.TopSage.com

其余的一些成员则是包含特定于某种操作系统的信息。如st_dev成员：在Windows操作系统中，它包含作为从ASCII字母A的偏移量来表示的驱动器信息（这就是我为什么在这个例子中使用字母A的原因，它给出驱动器的信息）。但是在Unix系统中，它就具有不同的意义；把它返回的值传给ustat系统调用将获得文件系统的名字。

如果需要更多的有关一个文件的信息，最好的办法就是从你的操作系统中获取。标准C的系统调用是围绕着Unix操作系统的，因此它们在Unix系统中常常具有更多有用的信息（并且有很多其他的系统调用可以与它们一起使用）。如果你使用的不是Unix系统，在你的操作系统的开发环境中有很多私有的库能够提供更详细的信息。

10.7 文件复制

问题

如何用一种可移植的办法来复制文件，例如，不使用操作系统特殊的API。

解决方案

使用<fstream>中C++文件流来把数据从一个流复制到另一个流。示例10-9给出了一个简单关于一个缓存流复制的例子。

示例10-9 复制文件

```
#include <iostream>
#include <fstream>

const static int BUF_SIZE = 4096;

using std::ios_base;

int main(int argc, char** argv) {

   std::ifstream in(argv[1],
      ios_base::in | ios_base::binary);  // Use binary mode so we can
   std::ofstream out(argv[2],            // handle all kinds of file
      ios_base::out | ios_base::binary); // content.

   // Make sure the streams opened okay...

   char buf[BUF_SIZE];

   do {
      in.read(&buf[0], BUF_SIZE);       // Read at most n bytes into
      out.write(&buf[0], in.gcount()); // buf, then write the buf to
   } while (in.gcount() > 0);           // the output.

   // Check streams for problems...

   in.close();
   out.close();
}
```

www.TopSage.com

讨论

从一个流读进数据然后再把这些数据写入另一个流从而完成复制一个文件，这种复制方式是简单的。但是C++流库是一个巨大的库，而且在这个库中有很多方法可以用来读和写，因此你应该对这个库有所了解以避免使得这些读和写操作性能不好。

示例10-9中的程序由于输入和输出都运用了缓存，它运行得很快。这些读和写函数都是一次操作在整个缓存上，而不是每一次只操作在一个字符上，通过成块地从一个流中读然后成块地写到另一个流。并且它也不做任何的对数据的格式化的工作如左移或右移操作等，这就使得它的运行非常快。另外，由于这个流工作在二进制模式上，文件结束符EOF也能被不间断地读和写。依赖于你使用的硬件和操作系统等，你将会得到不同的缓存大小。你需要做一些试验来为你的系统得到最好的参数。

但是还不止这些。所有的C++流在读和写时都已经用来缓存数据，因此示例10-9中事实上是双缓冲的。这个输入流有它自己的内部流缓冲，这个缓冲存储着那些从数据源中读进来的字符，而这些字符还没有被read、操作符<<、getc或者任何别的成员函数获取，而输出流也有它自己的内部缓冲，这些缓冲存储着这些已写入这个流但还没有写到目标地址（在这个例子中，ofstream流的目标是一个文件，也可以是一个字符串、一个网络连接或者别的东西）上的数据。因此，最好的办法就是让这些缓冲能够正确地交换数据。你可以使用操作符<<，它这时的功能不同于平常你把它运用在一个流缓冲时。例如，不使用示例10-9中的do/while循环，你可以如下这样做：

```
out << in.rdbuf();
```

不要只代替循环体中的语句，应该是用这个单一的语句来代替整个循环。它看上去有点怪异，因为典型地，操作符<<的意思是把它右边的数据送到左边的流中去，但是请忍受一下，一会你就会明白了。rdbuf从这个输入流中返回一个缓冲，并且这个带一个流缓冲作为它右边的参数操作符<<的实现一次从这个输出缓冲中读一个字符，然后把这个字符写入输出缓冲。当这个输出缓冲为空时，它知道这时不得不使用真正的数据源中的数据来填充它自己，而这个操作符<<则对这个填充一概不知。

示例10-9说明了如何亲自来复制一个文件中的内容，但是你的操作系统是可以管理文件系统的，包含复制文件等，因此为什么不让你的操作系统来做这项工作呢？在大部分情况下，这个问题的直接答案就是直接调用操作系统的API，当然这样的话，程序就是不可移植的。Boost文件系统通过提供copy_file函数来调用很多操作系统特有的函数，这个函数将依赖于你编译的操作系统平台来调用相应的操作系统函数。示例10-10包含了一个简单的程序来从一个地方复制一个文件到另一个地方。

示例10-10 使用Boost库来复制一个文件

```
#include <iostream>
#include <string>
```

www.TopSage.com

```
#include <boost/filesystem/operations.hpp>
#include <boost/filesystem/fstream.hpp>

using namespace std;
using namespace boost::filesystem;

int main(int argc, char** argv) {

   // Parameter checking...

   try {
       // Turn the args into absolute paths using native formatting
       path src = complete(path(argv[1], native));
       path dst = complete(path(argv[2], native));

       copy_file(src, dst);
   }
   catch (exception& e) {
      cerr << e.what() << endl;
   }

   return(EXIT_SUCCESS);
}
```

这是一个小程序，但是还是有些知识需要解释一下，因为这一章中的其他节也使用了Boost文件系统类库。作为开始，这个Boost文件系统类库的中心部件是path类，它以一种不依赖于操作系统的方式代表一个文件或目录的路径。你可以使用一个可移植或特定于某种操作系统的字符串来创建一个path对象。示例10-10中，我用程序外输入的参数来创建一个path对象（传入complete函数中的参数）。

```
path src = complete(path(argv[1], native));
```

第一个参数是这个路径的文本，如“tmp\\foo.txt”，而第二个参数接受这个字符串的函数名，并且这个函数返回一个布尔型的值来根据某种规则检验这个路径的合法性。这个native函数也就是意味着使用某种特殊的操作系统的格式来验证。示例10-10中我使用它是因为这个传入的参数是由某个人从命令行传入的，它将会是某种特殊操作系统的格式。有很多函数可以用来验证一个文件和目录路径的合法性，这些函数都是自解释的：ortable_posix_name、windows_name、portable_name、portable_directory_name、portable_file_name和no_check。更详细的请参考相应的文档。

complete函数使用当前使用的工作目录和你传入的相对路径来组合成一个绝对路径。因此，我可以这样来给某个源文件创建一个绝对路径：

```
path src = complete(path("tmp\\foo.txt", native));
```

当第一个参数就是一个绝对路径时，complete通过使用这个给定的路径而不是尽量把这个路径与当前的工作目录合并。换句话说，从当前目录“c:\myprograms”调用的以下代码将忽略这个当前目录，因为这个给定的目录已经就是全路径了。

```
path src = complete(path("c:\\windows\\garbage.txt", native));
```

如果前置条件没有满足的话，大部分的Boost文件系统函数将抛出一个异常。相应的文档

www.TopSage.com

中有详细的信息，但是一个很好的例子就是copy_file函数。在这个函数复制前这个文件必须存在，因此如果这个源文件不存在的话，这个操作将不能成功；因此这个copy_file函数将会抛出一个异常。就像我在示例10-10中那样捕获这个异常，你将获得这个错误的描述信息。

10.8 删除或重命名文件

问题

如何删除或重命名一个文件，并且你需要这些操作是可移值的，例如，不使用特定于某个操作系统的API?

解决方案

定义在<cstdio>中的标准的C函数remove和rename就可以完成这个工作。请看示例10-11中有关它们的一般描述。

示例10-11 删除一个文件

```
#include <iostream>
#include <cstdio>
#include <cerrno>

using namespace std;

int main(int argc, char** argv) {

   if (argc != 2) {
      cerr << "You must supply a file name to remove." << endl;
      return(EXIT_FAILURE);
   }

   if (remove(argv[1]) == -1) {  // remove() returns -1 on error
      cerr << "Error: " << strerror(errno) << endl;
      return(EXIT_FAILURE);
   }
   else {
      cout << "File '" << argv[1] << "' removed." << endl;
   }
}
```

讨论

这些系统调用是很容易使用的：你可以用你需要删除或重命名的文件名作为参数来调用它们。如果出错的话，它的返回值就是一个非零的整数并且errno被设置为某个特定的错误数。你可以使用strerror和perror（定义在<cstdio>）来打印出预先定义好的错误信息。

为了重命名一个文件，你可以用rename系统调用来代替示例10-11中的remove系统调用，代码如下所示：

www.TopSage.com

大家网 TopSage.com

```
    if (rename(argv[1], argv[2])) {
       cerr << "Error: " << strerror(errno) << endl;
       return(EXIT_FAILURE);
    }
```

Boost文件系统类库也提供了方法来删除或重命名一个文件。示例10-12说明了一个简短的例子来删除一个文件（或者是一个目录，请看这个例子后面的讨论部分）。

示例 10-12 使用 Boost 来删除一个文件

```
#include <iostream>
#include <string>
#include <boost/filesystem/operations.hpp>
#include <boost/filesystem/fstream.hpp>

using namespace std;
using namespace boost::filesystem;

int main(int argc, char** argv) {

   // Do parameter checking...

   try {
      path p = complete(path(argv[1], native));
      remove(p);
   }
   catch (exception& e) {
      cerr << e.what() << endl;
   }
   return(EXIT_SUCCESS);
}
```

示例 10-12 中的重要部分就是 `remove` 函数。使用一个指向一个文件或一个空目录来作为合法的路径参数来调用它，而这个作为参数的文件或目录就是要被删除的。关于这个 `path` 类和 `complete` 函数的解释，它们都是Boost文件系统类库的一部分，请参考10.7节的讨论和10.11节中的关于如何删除一个目录和这个目录包含的所有文件的例子。

重命名一个文件或目录是相同的。使用如下的代码来代替示例10-12代码中的try代码块：

```
    path src = complete(path(argv[1], native));
    path dst = complete(path(argv[2], native));
    rename(src, dst);
```

这将会把 `src` 重命名为 `dst`，只要它们都是一个合法的路径名。`src` 和 `dst` 不必具有相同的根目录，并且在这种情况下，这个rename函数能够把这个文件或目录移到新的目录中，只要 `dst` 确实存在就可以。

请参考 10.7 节。

10.9 创建一个临时文件名和文件

问题

如何临时地存储一些数据到磁盘上，并且你不想自己来实现这个产生唯一文件名的函数？

www.TopSage.com

解决方案

使用声明在<cstdio>中的tmpfile或tmpnam函数。tmpfile函数返回一个FILE*指针，它是这个为了输出而打开的文件指针，并且tmpnam产生一个你可以打开的唯一文件名。示例10-13说明了是如何使用tmpfile的。

示例10-13 创建一个临时文件

```
#include <iostream>
#include <cstdio>

using namespace std;

int main() {

   FILE* pf = NULL;
   char buf[256];

   pf = tmpfile();  // Create and open a temp file

   if (pf) {
      fputs("This is a temp file", pf);  // Write some data to it
   }

   fseek(pf, 5, SEEK_SET);  // Reset the file position
   fgets(buf, 255, pf);     // Read a string from it
   fclose(pf);

   cout << buf << '\n';
}
```

讨论

有两种办法可以用来创建一个临时文件；示例10-13说明了第一种。这个声明在<cstdio>中的tmpfile函数不带任何参数，并且成功后返回一个FILE*指针，不成功的话就返回一个NULL。这个FILE*类型就是你可以使用它作为C的输入输出函数（如fread、fwrite、fgets、fputs等）的参数。tmpfile使用“wb+”的方式打开一个临时文件，也就是你可以使用二进制模式向这个文件写或从这个文件读（例如，它的字符在读的时候是没有解释的）。当你的程序正常结束的话，这个由tmpfile函数创建的临时文件就会自动删除掉。

它能不能满足你的要求，这依赖于你的需求。你也许注意到了tmpfile函数不会给你一个文件名，你如何来把它传给另外一个程序呢？你不能，你将不得不使用另外一个函数：tmpnam。

tmpnam事实上不会产生一个临时文件，它只是创建一个唯一的文件名，你可以使用这个文件名来打开一个文件。tmpnam带一个char*类型的参数并返回一个char*类型的值。你可以使用一个指向一个字符缓存的指针来传递它（这个缓存的大小不得比定义在<cstdio>中的宏L_tmpnam的值小），tmpnam将复制这个临时的名字，并且返回一个指向这个缓存的指针。如果你传递一个NULL值的话，tmpfile将返回一个指向一个

www.TopSage.com

包含文件名的静态缓存的指针，也就意味着随后的tmpnam调用将会改写它（请参考示例10-14）。

示例10-14 创建一个临时文件

```
#include <iostream>
#include <fstream>
#include <cstdio>
#include <string>

using namespace std;

int main() {

   char* pFileName = NULL;

   pFileName = tmpnam(NULL);
   // Right here is where another program may get the same temp
   // filename.

   if (!pFileName) {
       cerr << "Couldn't create temp file name.\n";
       return(EXIT_FAILURE);
   }

   cout << "The temp file name is: " << pFileName << '\n';

   ofstream of(pFileName);

   if (of) {
      of << "Here is some temp data.";
      of.close();
   }

   ifstream ifs(pFileName);
   string s;

   if (ifs) {
      ifs >> s;
      cout << "Just read in \"" << s << "\"\n";
      ifs.close( );
   }
}
```

但是这里还是有些关于tmpnam函数的重要信息。它有一个竞态条件：如果一个进程调用tmpname并且另一个进程在前面这个进程打开这个文件之前调用tmpname的话，这两个进程可能会产生同样的文件名。这就非常不好，有两个方面的原因。第一，一个恶意的程序就能够攫取这个临时文件中的数据；第二，一个可信任的程序也可能得到这个相同的文件名并且销毁或删除这个临时文件的数据。

10.10 创建目录

问题

如何创建一个目录，并且需要使用可移植的方式来创建，例如，不使用特定于某个操作系统的API？

www.TopSage.com

解决方案

在大部分平台上，你都能使用mkdir系统调用，这个系统调用适用于大部分的C编译器。在不同的操作系统上，它有不同的形式，但是没关系，你可以使用它来创建一个新的目录。在C++中则没有标准的可移植的办法来创建一个目录。请参考示例10-15。

示例10-15 创建一个目录

```
#include <iostream>
#include <direct.h>
#include <cstring>
#include <errno.h>
#include <cstdlib>

using namespace std;

int main(int argc, char** argv) {

   if (argc < 2) {
      cerr << "Usage: " << argv[0] << " [new dir name]\n";
      return(EXIT_FAILURE);
   }

   if (mkdir(argv[1]) == -1) {  // Create the directory
      cerr << "Error: " << strerror(errno);
      return(EXIT_FAILURE);
   }
}
```

讨论

在不同的操作系统上这个系统调用还是有些不同的，但是你还是可以使用它来创建你的目录。mkdir的变体在大部分系统中都有支持，因此创建一个新目录也就是需要你知道在这个系统中哪个头文件中包含这个系统调用和这个系统调用的签名是什么等。

示例10-15在Windows上能工作正常，但在Unix上则不能。在Windows平台上，mkdir声明于<direct.h>头文件中，它带一个参数（目录名），出错则返回-1并且设置这个errno为某个特定的错误数。你可以调用strerror或perro来得到预先定义好的错误信息。

在Unix系统中，mkdir则声明在<sys/stat.h>文件中，而它的函数签名则有点不同。错误语义和Windows系统则相同，但是它带有两个参数，并且第二个参数指定了这个新目录上的权限。你可以使用传统的chmod格式来指定相应的权限（请参考chmod的帮助信息），例如，0777意味着所有者、所有者所在的组，以及别的用户具有读、写和执行权限。因此，你可以如下在Unix系统中调用这个系统调用：

```
#include <iostream>
#include <sys/types.h>
#include <sys/stat.h>
#include <cstring>
#include <errno.h>
#include <cstdlib>

using namespace std;
```

www.TopSage.com

```
int main(int argc, char** argv) {

   if (argc < 2) {
      cerr << "Usage: " << argv[0] << " [new dir name]\n";
      return(EXIT_FAILURE);
   }

   if (mkdir(argv[1], 0777) == -1) {  // Create the directory
      cerr << "Error: " << strerror(errno);
      return(EXIT_FAILURE);
   }
}
```

如果你需要写一个可移植的程序，并且你不想老使用#ifdefs条件编译的话，你就应该考虑使用Boost文件系统类库。你可以调用create_directory函数来创建一个新目录，如示例10-16中，它包含一个简短的程序来创建一个新目录。

示例10-16 使用Boost库来创建一个新的目录

```
#include <iostream>
#include <string>
#include <cstdlib>
#include <boost/filesystem/operations.hpp>
#include <boost/filesystem/fstream.hpp>

using namespace std;
using namespace boost::filesystem;

int main(int argc, char** argv) {

   // Parameter checking...

   try {
      path p = complete(path(argv[1], native));
      create_directory(p);
   }
   catch (exception& e) {
      cerr << e.what(  ) << endl;
   }

  return(EXIT_SUCCESS);
}
```

这个create_directory创建一个你作为参数传给它的目录。如果这个目录已经存在的话，一个filesystem_error的异常就会抛出（这个异常继承自标准的异常类）。请参考10.7节关于path类和complete函数的解释以及10.11节关于如何删除一个目录和它包含的所有文件。如果你不用考虑移植性的话，你就可以考虑使用你的特定操作系统上的API，它可能提供了更好的适应性。

请参考10.12节。

10.11 删除目录

问题

如何删除一个目录，并且你需要使用一种可移植的方法来删除它，例如，不使用特定于某种操作系统的API？

www.TopSage.com

大家网 TopSage.com

解决方案

在大部分平台上，你都可以使用rmdir系统调用来实现删除一个目录的功能，并且在大部分平台上都提供这个系统。C++则没有这样一个标准的可移植的办法来删除一个目录。rmdir在不同的操作系统上还是有所不同的，但是不管如何，你还是可以使用它来删除一个目录。示例10-17提供了一个简短的程序来删除一个目录。

示例10-17 删除一个目录

```
#include <iostream>
#include <direct.h>

using namespace std;

int main(int argc, char** argv) {

   if (argc < 2) {
      cerr << "Usage: " << argv[0] << " [dir name]" << endl;
      return(EXIT_FAILURE);
   }

   if (rmdir(argv[1]) == -1) {  // Remove the directory
      cerr << "Error: " << strerror(errno) << endl;;
      return(EXIT_FAILURE);
   }
}
```

讨论

这个rmdir系统调用在大部分操作系统上的签名都是相同的，但是声明它的头文可能不同。在Windwos系统上，它声明在<direct.h>中，而在Unix系统上，它声明在<unistd.h>文件中。它带一个参数(目录名)，如果出错则返回-1，并且设置这个errno为某个相应的错误编号。你可以使用strerror或perror来得到预先定义好的错误信息。

如果rmdir要删除的目标目录是空的则返回一个错误。为了列举出这个目录中的内容从而也确保把它们删除掉，请参考10.12节。

如果你为可移植性考虑，并且不想自己使用#ifdefs等条件编译来围绕着各种特定于操作系统的目录功能，就应该考虑使用Boost Filesystem库。这个库使用了路径的概念来引用目录或文件，并且这个path可以与一个简单的remove函数一起适用来删除目录。

示例10-18中的removeRecurse函数递归地删除一个目录和它的所有内容。这个例子中最重要的就是remove函数（它是boost::filesystem::remove而不是标准类库函数)。它带一个路径名作为参数，并且如果它是一个文件或一个空目录的话就删除它，但它不删除一个还包含文件的目录。

www.TopSage.com

示例 10-18 使用 Boost 库来删除一个目录

```
#include <iostream>
#include <string>
#include <cstdlib>
#include <boost/filesystem/operations.hpp>
#include <boost/filesystem/fstream.hpp>

using namespace std;
using namespace boost::filesystem;

void removeRecurse(const path& p) {

   // First, remove the contents of the directory
   directory_iterator end;
   for (directory_iterator it(p);
   it != end; ++it) {

 if (is_directory(*it)) {
    removeRecurse(*it);
 }
 else {
    remove(*it);
 }
   }
   // Then, remove the directory itself
   remove(p);
}

int main(int argc, char** argv) {

   if (argc != 2) {
      cerr << "Usage: " << argv[0] << " [dir name]\n";
      return(EXIT_FAILURE);
   }

   path thePath = system_complete(path(argv[1], native));

   if (!exists(thePath)) {
      cerr << "Error: the directory " << thePath.string()
           << " does not exist.\n";
      return(EXIT_FAILURE);
   }

   try {
      removeRecurse(thePath);
   }
   catch (exception& e) {
      cerr << e.what() << endl;
      return(EXIT_FAILURE);
   }
   return(EXIT_SUCCESS);
}
```

这些列举出一个目录中的内容的代码还是需要一些解释的，并且这是 10.12 节的主题。

这个Boost文件类库是方便的，但是请记住这个Boost库不是正式的标准，并且它不能保证在任何地方都可以运行。如果你检查这个Boost文件系统类库源代码，你必须注意到它被编译成目标平台的代码。因此如果可移植性不用考虑的话，你还是应考虑一下你的操作系统文件系统的 API，它或许可以提供更多的灵活性。

请参考 10.12 节

www.TopSage.com

10.12 读取目录的内容

问题

如何读取一个目录中内容，最有可能的就是需要对它的每一个文件或它的每一个子目录做某种操作？

解决方案

为了实现一个可移植的方法，你可以使用Boost文件系统类库中的类和函数。它提供了很多方便的操作文件的方法，如一个可移植的路径表示，目录迭代器以及许多用于重命名、删除和复制文件的函数。示例 10-19 说明了如何使用这些方法。

示例 10-19 读取一个目录

```
#include <iostream>
#include <boost/filesystem/operations.hpp>
#include <boost/filesystem/fstream.hpp>

using namespace boost::filesystem;

int main(int argc, char** argv) {

   if (argc < 2) {
      std::cerr << "Usage: " << argv[0] << " [dir name]\n";
      return(EXIT_FAILURE);
   }

   path fullPath =    // Create the full, absolute path name
system_complete(path(argv[1], native));

   if (!exists(fullPath)) {
      std::cerr << "Error: the directory " << fullPath.string()
                << " does not exist.\n";
      return(EXIT_FAILURE);
   }

   if (!is_directory(fullPath)) {
      std::cout << fullPath.string() << " is not a directory!\n";
      return(EXIT_SUCCESS);
   }

   directory_iterator end;
   for (directory_iterator it(fullPath);
       it != end; ++it) {       // Iterate through each
                                // element in the dir,
      std::cout << it->leaf(); // almost as you would
      if (is_directory(*it))   // an STL container
         std::cout << " (dir)";
         std::cout << '\n';
   }

   return(EXIT_SUCCESS);
}
```

讨论

就像创建或删除目录（10.10节和10.11节）那样，读取一个目录的内容也没有一个可移植的办法。为了让你的C++程序变得容易，Boost项目中的文件系统类库提供了一序列

www.TopSage.com

的操作在文件和目录上的可移植的方法。当然它也提供了一些别的方法，你可以参考这一章的其他节或者 Boost 网站中有关文件系统的页面。

示例 10-19 是一个简单的目录列举程序（就像 Unix 上的 ls 或 MS-DOS 上的 dir 程序一样）。首先，它从传给它的参数中构建一个完整的路径，如下所示：

```
path fullPath = complete(path(argv[1], native));
```

一个路径的数据类型是 path。文件系统的函数都操作在这个类型上，并且它很容易通过调用 path::string 转化成字符串。一旦这个路径组合完成了，这个程序就可以检查它是否存在（通过 exists），然后通过另外一个函数 is_directory 来检查它是否是一个目录。如果是的话，那么就可以继续下面的工作来列举这个目录中所有的内容。

这个在文件系统中称为 directory_iterator 的类使用标准的 iterator 语义，就像标准的容器那样，让你可以使用一个 iterator 来枚举目录中的元素。然而不像标准的容器，这里没有 end 成员函数可以让你调用来代表指向最后这个元素后的那个元素（如 vector<T>::end）。相反，如果你使用默认的构造函数来创建一个 directory_iterator，它代表一个结束标志并且你可以使用它来比较，从而决定是否目录中没有元素了。如下所示：

```
directory_iterator end;
```

然后你就可以从你的 path 上创建一个 iterator 并且与这个 end 相比较，如下所示：

```
for (directory_iterator it(fullPath);
it != end; ++it) {
   // do whatever you want to *it
   std::cout << it->leaf();
}
```

这个 leaf 成员函数返回一个字符串，这个字符串代表一个 path 的元素，并且如果它不是这个完整路径本身的话，就正是你需要的。

如果你需要写一个可移植的方案，但由于某种原因你又不能使用 Boost 库，你就可以参考一下 Boost 源代码。它包含 #ifdefs 条件编译来处理各种操作系统接口环境和特殊的路径，如驱动器字母或驱动器名字。

请参考 10.10 节和 10.11 节。

10.13 从字符串中提取文件扩展名

问题

给定一个文件名或一个完整的路径名，如何获取文件的扩展名？扩展名是指文件名中点号后面的那部分。例如，在文件名 *src.cpp*、*Window.class* 和 *Resume.doc* 中，它们的扩展名分别是 *.cpp*、*.class* 和 *.doc*。

www.TopSage.com

解决方案

把这个文件名或路径名转换成一个字符串，然后使用rfind成员函数来定位最后一个点号，并且返回它后面的字符。示例 10-20 说明了是如何处理的。

示例 10-20 从文件名中获取扩展名

```
#include <iostream>
#include <string>

using std::string;

string getFileExt(const string& s) {

   size_t i = s.rfind('.', s.length());
   if (i != string::npos) {
      return(s.substr(i+1, s.length() - i));
   }

   return("");
}

int main(int argc, char** argv) {

   string path = argv[1];

   std::cout << "The extension is \"" << getFileExt(path) << "\"\n";
}
```

讨论

为了从一个文件名中得到它的扩展名，你需要找到最后一个点号所在的位置并且返回这个点号右边的字符。定义在<string>中的string字符串类包含了处理这两个功能的成员函数：rfind和substr。

rfind成员函数向后查找你作为第一个参数传给它的值(在这个例子中是字符)，它从这个函数的第二个参数指定的下标开始查找，并且返回找到的下标。如果不能找到的话，rfind函数就返回string::npos。substr也带两个参数。第一个就是需要复制的第一个元素的下标，而第二个就是要复制的字符个数。

这个标准的 string 类包含一序列的查找成员函数。更多的关于字符串查找的讨论请参考 4.9 节。

请参考 4.9 节和 10.12 节。

10.14 从完整路径中提取文件名

问题

有一个完整路径名的文件，例如，*d:\apps\src\foo.c*，如何从这个完整路径名中获取文件的名字*foo.c*?

www.TopSage.com

大家网 TopSage.com

解决方案

使用与前几节中相同的技术rfind和substr来从这个完整路径名中获取这个文件名。示例10-21说明了是如何实现的。

示例10-21 从一个完整的路径名中获取文件名

```
#include <iostream>
#include <string>

using std::string;

string getFileName(const string& s) {

   char sep = '/';

#ifdef _WIN32
   sep = '\\';
#endif

   size_t i = s.rfind(sep, s.length());
   if (i != string::npos) {
     return(s.substr(i+1, s.length() - i));
   }

   return("");
}

int main(int argc, char** argv) {

   string path = argv[1];

   std::cout << "The file name is \"" << getFileName(path) << "\"\n";
}
```

讨论

对于rfind和substr是如何工作的请参考前面几节。示例10-21中唯一值得说明的就是，或许你已经意识到了，Windows系统使用反斜线符号而不是斜线符号，因此我加上了一个#ifdef来条件设置这个路径分割符。

Boost文件系统类库中的path类可以得到完整路径名的最后一部份，它可能是一个文件或目录名，使用path::leaf成员函数很容易得到它。示例10-22给出了一个简单的例子来打印出这个路径指向的是文件还是目录。

示例10-22 从一个完整路径中获取文件名

```
#include <iostream>
#include <cstdlib>
#include <boost/filesystem/operations.hpp>

using namespace std;
using namespace boost::filesystem;

int main(int argc, char** argv) {

   // Parameter checking...
```

www.TopSage.com

```
    try {
       path p = complete(path(argv[1], native));
       cout << p.leaf() << " is a "
            << (is_directory(p) ? "directory" : "file") << endl;
    }
    catch (exception& e) {
       cerr << e.what() << endl;
    }

  return(EXIT_SUCCESS);
}
```

更多的关于path类的信息请看10.7节中的讨论。

请参考10.15节。

10.15 从完整路径和文件名中提取路径

问题

有一个文件名的完整路径，例如d:\apps\src\foo.c，如何从这个完整路径中取出路径名d:\apps\src?

解决方案

使用和前两节中相同的技术，通过调用rfind和substr来从这个完整的路径中获取到你需要的路径名。示例10-23给出了一个简单的例子程序。

示例10-23 从一个完整的路径和文件名中获取路径名

```
#include <iostream>
#include <string>

using std::string;

string getPathName(const string& s) {

   char sep = '/';

#ifdef _WIN32
   sep = '\\';
#endif

   size_t i = s.rfind(sep, s.length());
   if (i != string::npos) {
     return(s.substr(0, i));
   }

   return("");
}

int main(int argc, char** argv) {

   string path = argv[1];

   std::cout << "The path name is \"" << getPathName(path) << "\"\n";
}
```

www.TopSage.com

讨论

示例10-23是简单的，特别是如果你已经学过了前几节的内容时，因此就不需要更多的解释了。然而，就像其他几节那样，Boost文件系统类库提供了branch_path函数来获取除了最后的那个文件名之外的别的东西。示例10-24说明是如何做的。

示例 10-24 获取路径名

```
#include <iostream>
#include <cstdlib>
#include <boost/filesystem/operations.hpp>

using namespace std;
using namespace boost::filesystem;

int main(int argc, char** argv) {

   // Parameter checking...

   try {
      path p = complete(path(argv[1], native));
      cout << p.branch_path().string() << endl;
   }
   catch (exception& e) {
      cerr << e.what() << endl;
   }
  return(EXIT_SUCCESS);
}
```

示例 10-24 的输出如下：

```
D:\src\ccb\c10>bin\GetPathBoost.exe c:\windows\system32\1033
c:/windows/system32
```

请参考 10.13 节和 10.14 节。

10.16 替换文件扩展名

问题

给定一个文件名或者一个路径和文件名，如何替换这个文件的扩展名？例如，你有一个名为thesis.tex的文件，你想把它转换成thesis.txt。

解决方案

使用字符串类的rfind和replace成员函数来找到文件的扩展名并替换它。示例10-25说明如何这样做。

示例 10-25 替换文件的扩展名

```
#include <iostream>
#include <string>

using std::string;

void replaceExt(string& s, const string& newExt) {
```

www.TopSage.com

```
    string::size_type i = s.rfind('.', s.length());

    if (i != string::npos) {
      s.replace(i+1, newExt.length(), newExt);
    }
}

int main(int argc, char** argv) {

    string path = argv[1];

    replaceExt(path, "foobar");
    std::cout << "The new name is \"" << path << "\"\n";
}
```

讨论

这个解决方案和前几节中相同，但在这个例子中通过replace成员函数，用一个新的字符串来代替另一个字符串中的一部分。replace成员函数有三个参数。第一个参数是替换开始处的下标，第二个参数是要从目标字符串删除的字符个数，第三个参数就是将要替换目标字符串中被删除的字符的值。

请参考4.9节

10.17 将两个路径组合成一个路径

问题

有两个路径，如何把它们合并到一个单一的路径中？你可能首先有一个/usr/home/ryan的路径，然后你有第二个utils/compilers的路径，你想把它们合并成/usr/home/ryan/utils/compilers，不用担心第一个路径是否是以路径分割符结尾。

解决方案

把路径当成字符串并且使用这个追加操作符，操作符+=来把部分路径合并成一个完整路径。请看示例10-26。

示例10-26 合并路径

```
#include <iostream>
#include <string>

using std::string;

string pathAppend(const string& p1, const string& p2) {

    char sep = '/';
    string tmp = p1;

#ifdef _WIN32
  sep = '\\';
#endif

  if (p1[p1.length()] != sep) { // Need to add a
     tmp += sep; // path separator
```

www.TopSage.com

```
    return(tmp + p2);
  }
  else
    return(p1 + p2);
}

int main(int argc, char** argv) {

   string path = argv[1];

   std::cout << "Appending somedir\\anotherdir is \""
             << pathAppend(path, "somedir\\anotherdir") << "\"\n";
}
```

讨论

示例10-26中的代码使用字符串来代表路径，但是在这个path类中并没有为了合法性的附加检查并且这个路径仅使用它们能够移植的值。例如，这些路径是从用户获取的，你不用知道它们是否使用了正确的特定于某种操作系统的格式，或者它们是否包含了不合法的字符。

本章很多节中我都包含了那些使用Boost文件系统类库的例子，并且当它们与路径一起工作时，这种方法有很多好处。就像我在10.7节中讨论的那样，Boost文件系统类库包含了一个path类，它是文件或目录路径的一个可移植的表示。在这个文件系统类库中的操作大部分都能与path类对象一起工作，因此，这个path类能够处理从一个绝对路径和一个相对路径的合并（见示例10-27）。

示例10-27 使用Boost库来合并路径

```
#include <iostream>
#include <string>
#include <cstdlib>
#include <boost/filesystem/operations.hpp>
#include <boost/filesystem/fstream.hpp>

using namespace std;
using namespace boost::filesystem;

int main(int argc, char** argv) {

   // Parameter checking...

   try {
      // Compose a path from the two args
      path p1 = complete(path(argv[2], native),
                         path(argv[1], native));
      cout << p1.string() << endl;

      // Create a path with a base of the current dir
      path p2 = system_complete(path(argv[2], native));
      cout << p2.string() << endl;
   }
   catch (exception& e) {
      cerr << e.what() << endl;
   }

  return(EXIT_SUCCESS);
}
```

www.TopSage.com

大家网 TopSage.com

示例10-27中程序的输出如下：

```
D:\src\ccb\c10>bin\MakePathBoost.exe d:\temp some\other\dir
d:/temp/some/other/dir
D:/src/ccb/c10/some/other/dir
```

或：

```
D:\src\ccb\c10>bin\MakePathBoost.exe d:\temp c:\WINDOWS\system32
c:/WINDOWS/system32
c:/WINDOWS/system32
```

上面你能看到就是当有可能时complete和system_complete就合并路径，或者当合并没有意义时就返回一个绝对路径。例如，在上面第一个输出中，传给这个程序的第一个参数是一个绝对路径并且第二参数是一个相对路径。complete函数就把它们合并到一起并产生一个单一的绝对路径。第一个参数是一个相对路径，第二个参数是一个绝对路径，如果第一个参数已经是一个绝对路径的话，第二个参数就会被忽略。这就是为什么第二个输出中参数"d:\temp"被忽略的原因，因为第二个参数已经是一个绝对路径了。

system_complete仅带一个参数（在这个例子中是一个相对路径）并且把它追加到当前的工作目录来产生另外一个绝对路径。如果你传给它的参数已经是一个绝对路径，它将忽略当前的工作目录并简单地返回这个你给定的绝对路径。

这些路径在文件系统中并不能保证存在。你不得不显式测试这个path对象所代表的路径是否是一个合法的文件路径。例如，为了测试这些路径是否存在，你可以使用exists函数。

```
path p1 = complete(path(argv[2], native), path(argv[1], native));
if (exists(p1)) {
   // ...
```

你可以使用很多函数来获取一个路径的信息：is_dir-ectory、is_empty、file_size、last_write_time等等。更多的关于Boost文件系统的信息请参考*www.boost.org*。

请参考10.7节。

www.TopSage.com

第11章

科学与数学计算

11.0 概述

C++由于它的灵活性、可表达性和高效率，所以它也是一个非常适合科学和数学编程的语言。C++为数字处理编码的一个最大的优点就是它能帮助你避免冗余。

历史上，很多语言中的数字编码将不得不在各种不同的数字类型中重复同样的算法（如短整型、长整型、单精度型、双精度型和用户自定义的数字类型等）。通过模板，C++给这种冗余问题提供了解决方案。模板使得你可以写一个独立于数据类型的算法，通常这种技术也叫做泛型编程。

在数字处理编码中，C++也并不是没有缺点。与某些特殊的数学和科学编程语言相比，C++的最大的缺点就是它的标准类库只限制在与数字编程相关的算法和数据类型上的支持。它的标准类库中最大的缺点就是缺乏矩阵类型的支持并缺乏绝对精度的整型。

在这一章中，我将给你提供一些关于共同的数字编程问题的解决方案并说明了如何使用泛型编程技术来有效地实现一些数字编码。合适的情况下，我将向你推荐一些商业上友好许可的使用很广并且有着记录跟踪的开源类库。这一章将一节一节逐渐地介绍基本的泛型编程技术。

许多使用C++的编程人员由于模板和泛型编程的明显的复杂性仍然不相信模板和泛型编程。当模板刚引进C++语言时，它既没有被实现得很好，也没有被编程人员和编译器的实现者理解得很好。结果，许多编程人员（可能包括你）在C++中很多年都避免使用泛型编程，尽管这项技术已经成熟。

今天，作为一项强大的有用的编程模式，泛型编程已经被广泛接受，并且在大多数流行的编程语言中得到了支持。C++编译器技术得到了飞跃的发展，并且现代编译器都以更加标准化和更有效的方式处理模板。结果，现代的C++就是一个适合科学和数学计算应用的功能强大的语言。

www.TopSage.com

11.1 计算一个容器中元素的个数

问题

如何找出一个容器中元素的个数？

解决方案

通过使用size成员函数或定义在<algorithm>头文件中的distance函数，你就可以计算出一个容器中元素的个数。如示例11-1所示。

示例11-1 计算一个容器中元素的个数

```
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

int main() {
  vector<int> v;
  v.push_back(0);
  v.push_back(1);
  v.push_back(2);
  cout << v.size() << endl;
  cout << distance(v.begin(), v.end()) << endl;
}
```

示例11-1中程序的输出如下：

```
3
3
```

讨论

这个size成员函数返回一个标准容器中元素的个数，当这个容器对象可以访问时，它是一个最好的解决方案。在示例11-1中，我也说明了distance函数，因为当你需要写泛型代码时，它是通常与一对迭代器一起工作的方法。当你使用迭代器时，常常不需要访问这个容器的类型或者它的成员函数。

就像大部分STL算法一样，这个distance函数事实上是一个模板函数。由于这个模板参数的类型可以被编译器自动地从一个函数参数中推演出来，所以你不必不得不显式地把一个模板参数传给它。当然，如果你愿意的话，你也可以显式地写出这个模板参数，如下所示：

```
cout << distance<vector<int>::iterator>(v.begin(), v.end()) << endl;
```

这个diatance函数的性能依赖于使用的迭代器的类型。如果这个迭代器是一个随机访问的迭代器，那么它就只用花一个常数时间；否则，就是一个线形时间（迭代器的概念请参考7.1节）。

请参考15.1节。

www.TopSage.com

11.2 查找最大或最小值

问题

如何找出一个容器中的最大值或者最小值？

解决方案

通过使用定义在头文件<algorithm>中的函数max_element和min_element，示例11-2中说明了如何在一个容器中找到它的最大和最小元素。这两个函数分别返回指向与这个最大或最小值相对应的第一次匹配的元素的迭代器。

示例11-2 从一个容器中查找它的最小或最大元素

```
#include <algorithm>
#include <vector>
#include <iostream>

using namespace std;

int getMaxInt(vector<int>& v) {
  return *max_element(v.begin(), v.end());
}

int getMinInt(vector<int>& v) {
  return *min_element(v.begin(  ), v.end(  ));
}

int main() {
  vector<int> v;
  for (int i=10; i < 20; ++i) v.push_back(i);
  cout << "min integer = " << getMinInt(v) << endl;
  cout << "max integer = " << getMaxInt(v) << endl;
}
```

示例11-2中程序的输出如下：

```
    min integer = 10
    max integer = 19
```

讨论

你或许注意到间接引用了从这个调用min_element和max_element返回的值。这是因为这两个函数返回迭代器而不是事实上的值，因此其结果不得不被间接引用。你或许觉得不得不间接引用这个返回类型有点不方便，但这样避免了不必要的值复制。当这个返回值的复制操作很昂贵时，这一点很重要（如一个大的字符串）。

标准类库中提供的泛型算法显然更有用，但是为这个从一个容器中得到最小和最大值写出你自己的泛型函数对你来说更重要。例如，假设你需要一个单一的函数，它通过修改这个引用参数而不是用一对这种值或者其他的结构来返回这个最小和最大值。示例11-3中说明了是如何做的。

www.TopSage.com

示例 11-3 返回最小值和最大值的泛型函数

```
#include <algorithm>
#include <vector>
#include <iostream>

using namespace std;

template<class Iter_T, class Value_T>
void computeMinAndMax(Iter_T first, Iter_T last, Value_T& min, Value_T& max)
{
  min = *min_element(first, last);
  max = *max_element(first, last);
}

int main() {
  vector<int> v;
  for (int i=10; i < 20; ++i) v.push_back(i);
  int min = -1;
  int max = -1;
  computeMinAndMax(v.begin(), v.end(), min, max);
  cout << "min integer = " << min << endl;
  cout << "max integer = " << max << endl;
}
```

在示例11-3中，我实现了一个computeMinAndMax函数模板，它带两个模板参数，一个是迭代器的类型，另一个是这个最小值和最大值的类型。因为这两个模板参数都是函数参数，C++编译器能够推演出它们各自的类型Iter_T和Value_T，就像我在11.1节中说明的那样。这使得我不用显式地指明这两个模板参数：

```
compute_min_max<vector<int>::iterator, int>(...)
```

min_element 和 max_element函数通过使用操作符<来比较由这个迭代器引用的值。这就意味着如果一个迭代器引用的类型不支持这个小于操作符比较，那么一个编译错误就会发生。然而，这个min_element和max_element函数也可使用一个用户定义的比较算符，如一个函数指针或一个函数对象。

算符

许多STL算法都接受用户自己定义的函数对象或函数指针。它们统称为算符。有时在一些文献中，这个函数对象术语与这个算符术语是可以互换的，但是我保留使用函数对象作为特别地引用重载了操作符()的类或结构的实例。

在这两种选择中，你应该使用哪种呢？在大部分情况下，函数对象更有效，因为大部分编译器都更容易内联一个函数对象。

另一个使用一个函数对象的原因就是它可以有状态。你可以传值给它的构造函数，它能够在它的字段中存储这个值来为以后使用。这就使得这个函数对象可以表达相等的概念，就像在其他编程语言中的闭包的概念一样。

最后，函数对象可以在别的函数或类内部定义。函数指针则不得不在一个命名空间中声明。

www.TopSage.com

min_element 和 max_element 特殊类型的算符是一个需要带这个由迭代器引用的类型的两个值，并且如果一个比另外一个小的话就返回一个布尔值。返回一个布尔值的算符称为一个断言。例如，考虑示例11-4中查找一序列用户自定义类型中的最大元素的情况。

示例 11-4 查找一个用户自定义类型中的最大元素

```
#include <algorithm>
#include <vector>
#include <iostream>

using namespace std;

struct ChessPlayer {
  ChessPlayer(const char* name, int rating)
    : name_(name), rating_(rating)
  { }
  const char* name_;
  int rating_;
};

struct IsWeakerPlayer {
  bool operator()(const ChessPlayer& x, const ChessPlayer& y) {
    return x.rating_ < y.rating_;
  }
};

int main()
{
  ChessPlayer kasparov("Garry Kasparov", 2805);
  ChessPlayer anand("Viswanathan Anand ", 2788);
  ChessPlayer topalov("Veselin Topalov", 2788);
  vector<ChessPlayer> v;
  v.push_back(kasparov);
  v.push_back(anand);
  v.push_back(topalov);
  cout << "the best player is ";
  cout << max_element(v.begin(), v.end(), IsWeakerPlayer(  ))->name_;
  cout << endl;
}
```

示例 11-4 中的程序输出如下：

```
    the best player is Garry Kasparov
```

在示例11-4我说明了如何与一个自定义的断言来一起提供max_element。这个断言是函数对象 IsWeakerPlayer。

另一个代替示例11-4中的用户定义的断言的方法是为ChessPlayer结构重写操作符<。这对于例子中的某些特殊情况将很好，但它假定了给这些玩家排序的重要方法是通过rating方法。可能根据名字来排序是更流行的。在这个例子中，由于排序方法的选择是任意的，所以我更愿意不去定义操作符 <。

11.3 计算总和与平均值

问题

如何计算一个容器中所有元素的总和，以及这些元素的平均值？

www.TopSage.com

解决方案

你可以使用来自头文件<numeric>中的accumulate函数来计算总和，并且用这个总和除以这个容器的大小size来得到它的平均值。示例11-5中说明了vector的这种用法。

示例11-5 计算一个容器中元素的总和以及平均值

```
#include <numeric>
#include <iostream>
#include <vector>

using namespace std;

int main() {
  vector<int> v;
  v.push_back(1);
  v.push_back(2);
  v.push_back(3);
  v.push_back(4);
  int sum = accumulate(v.begin(), v.end(), 0);
  double mean = double(sum) / v.size();
  cout << "sum = " << sum << endl;
  cout << "count = " << v.size() << endl;
  cout << "mean = " << mean << endl;
}
```

示例11-5中的程序输出如下：

```
sum = 10
count = 4
mean = 2.5
```

讨论

总的来说，accumulate函数给找到一个容器中所有元素的总和提供了最有效和最简单的方法。

尽管这一节有一个相对简单的方案，但是实现一个你自己的泛型函数来计算平均值也不是那么容易。示例11-6说明了一种实现这种泛型函数的方法：

示例11-6 一个计算平均值的泛型函数

```
template<class Iter_T>
double computeMean(Iter_T first, Iter_T last) {
  return static_cast<double>(accumulate(first, last, 0.0))
    / distance(first, last);
}
```

示例11-6中的computeMean函数对于大部分目的是足够的，但它也有一个限制：它不能与输入迭代器如istream_iterator一起工作。

一个istream_iterator让你可以简单地在数据上传递，因此你可以调用accumulate函数或者distance函数，但是当你每次调用这两个函数中的一个时，数据会变得不合法并且进一步的尝试枚举这些数据都将失败。示例11-7说明了如何在一个简单传递中计算一序列数字的平均值实现一个更一般的算法。

www.TopSage.com

istream_iterator 和 ostream_iterator

这个istream_iterator和ostream_iterator类模板是头文件<iterator>中的专用迭代器，它使得你可以把一个流当作简单传递的容器。

istream_iterator是一个包裹了一个输入流的迭代器，如cin或者ifstream，使得它可以被用作很多泛型函数的参数。ostream_iterato是一个输出迭代器，让你可以把一个输出流作为一个容器那样来使用。

使用istream_iterator和ostream_iterator是一个好的习惯，因为它们使得你写出可重用的代码更容易。

示例11-7 计算平均值的一个更一般的函数

```
#include <stdexcept>
#include <iostream>
#include <iterator>

using namespace std;

template<class Value_T, class Iter_T>
Value_T computeMean(Iter_T first, Iter_T last) {
  if (first == last) throw domain_error("mean is undefined");
  Value_T sum;
  int cnt = 0;
  while (first != last) {
    sum += *first++;
    ++cnt;
  }
  return sum / cnt;
}

int main() {
  cout << "please type in several integers separated by newlines" << endl;
  cout << "and terminated by an EOF character (i.e., Ctrl-Z)" << endl;
  double mean = computeMean<double>(
    istream_iterator<int>(cin),
    istream_iterator<int>( ));
  cout << "the mean is " << mean << endl;
}
```

当你写泛型代码时，你应该一直尽力使用最基本类型的迭代器。这就意味着无论何时，你都应该实现操作在一个输入上的简单传递的泛型算法。通过这种方法，你的泛型代码就不会仅限制在容器上，并且它也能使用在输入迭代器如istream_iterator上。更进一步，简单传递算法通常也是更加有效的。

示例11-7中，我决定实现的computeMean函数要求这个返回类型是作为模板参数传入的而不是从这个迭代器类型推演出来的，也许你对这一点觉得吃惊。这是因为为了统计，使用一个更高精度级别而不是容器中的数字级别来计算更为常用。例如，示例11-7中的代码以一个双精度值作为一序列的整型值的平均值返回。

www.TopSage.com

11.4 过滤掉给定范围之外的值

问题

如何忽略掉一个给定范围之外的值？

解决方案

使用定义在 <algorithm> 头文件中的 remove_copy_if 函数，如示例 11-8 所示。

示例 11-8 从一个序列中删除掉小于某个值的元素

```
#include <algorithm>
#include <vector>
#include <iostream>
#include <iterator>

using namespace std;

struct OutOfRange
{
  OutOfRange(int min, int max)
    : min_(min), max_(max)
  { }
  bool operator()(int x) {
    return (x < min_) || (x > max_);
  }
  int min_;
  int max_;
};

int main()
{
  vector<int> v;
  v.push_back(6);
  v.push_back(12);
  v.push_back(18);
  v.push_back(24);
  v.push_back(30);
  remove_copy_if(v.begin(), v.end(),
    ostream_iterator<int>(cout, "\n"), OutOfRange(10,25));
}
```

示例 11-8 中程序的输出如下：

```
    12
    18
    24
```

讨论

remove_copy_if 函数从一个容器中复制元素到另一个容器中（或者是输出迭代器），它忽略掉满足这个你提供的断言的元素（或许这个函数重命名为 copy_ignore_if 则更能反应它的功能）。然而，这个函数并不改变目标容器的大小。就像这个例子一样，如果由这个 remove_copy_if 复制的元素个数比目标容器的容量小的话，你将不得不调用 erase 成员函数来缩短这个目标容器。

www.TopSage.com

remove_copy_if函数需要一个一元断言(一个带一个参数的算符并且它返回一个布尔值)，当这个断言返回真值时，这个元素就不应该被复制。在示例11-8中，这个算符就是函数对象OutOfRange。这个OutOfRange的构造函数带一个上限和下限的范围，并且重载了操作符()。这个操作符()带一个整型参数，并且如果这个传入的参数小于这个上限或者比这个下限大的话就返回真值。

11.5 方差、标准差及其他统计函数计算

问题

如何对一序列的数字计算一个或多个通常的统计指标，如方差、标准差、偏差和峭度？

解决方案

你可以使用<numeric>头文件中的accumulate函数传入自定义的函数对象来计算简单的总和之外的多个有意义的统计功能。示例11-9中说明了该如何使用accumulate函数来计算几个重要的统计功能。

示例 11-9 统计函数

```
#include <numeric>
#include <cmath>
#include <algorithm>
#include <functional>
#include <vector>
#include <iostream>

using namespace std;

template<int N, class T>
T nthPower(T x) {
  T ret = x;
  for (int i=1; i < N; ++i) {
    ret *= x;
  }
  return ret;
}

template<class T, int N>
struct SumDiffNthPower {
  SumDiffNthPower(T x) : mean_(x) { };
  T operator()(T sum, T current) {
    return sum + nthPower<N>(current - mean_);
  }
  T mean_;
};

template<class T, int N, class Iter_T>
T nthMoment(Iter_T first, Iter_T last, T mean)  {
  size_t cnt = distance(first, last);
  return accumulate(first, last, T(), SumDiffNthPower<T, N>(mean)) / cnt;
}

template<class T, class Iter_T>
T computeVariance(Iter_T first, Iter_T last, T mean) {
  return nthMoment<T, 2>(first, last, mean);
}
```

www.TopSage.com

大家网 TopSage.com

```
template<class T, class Iter_T>
T computeStdDev(Iter_T first, Iter_T last, T mean) {
  return sqrt(computeVariance(first, last, mean));
}

template<class T, class Iter_T>
T computeSkew(Iter_T begin, Iter_T end, T mean) {
  T m3 = nthMoment<T, 3>(begin, end, mean);
  T m2 = nthMoment<T, 2>(begin, end, mean);
  return m3 / (m2 * sqrt(m2));
}

template<class T, class Iter_T>
T computeKurtosisExcess(Iter_T begin, Iter_T end, T mean) {
  T m4 = nthMoment<T, 4>(begin, end, mean);
  T m2 = nthMoment<T, 2>(begin, end, mean);
  return m4 / (m2 * m2) - 3;
}

template<class T, class Iter_T>
void computeStats(Iter_T first, Iter_T last, T& sum, T& mean,
  T& var, T& std_dev, T& skew, T& kurt)
{
  size_t cnt = distance(first, last);
  sum = accumulate(first, last, T());
  mean = sum / cnt;
  var = computeVariance(first, last, mean);
  std_dev = sqrt(var);
  skew = computeSkew(first, last, mean);
  kurt = computeKurtosisExcess(first, last, mean);
}

int main()
{
  vector<int> v;
  v.push_back(2);
  v.push_back(4);
  v.push_back(8);
  v.push_back(10);
  v.push_back(99);
  v.push_back(1);
  double sum, mean, var, dev, skew, kurt;
  computeStats(v.begin(), v.end(), sum, mean, var, dev, skew, kurt);
  cout << "count = " << v.size() << "\n";
  cout << "sum = " << sum << "\n";
  cout << "mean = " << mean << "\n";
  cout << "variance = " << var << "\n";
  cout << "standard deviation = " << dev << "\n";
  cout << "skew = " << skew << "\n";
  cout << "kurtosis excess = " << kurt << "\n";
  cout << endl;
}
```

示例11-9中程序的输出如下所示：

```
count = 6
sum = 124
mean = 20.6667
variance = 1237.22
standard deviation = 35.1742
skew = 1.75664
kurtosis excess = 1.14171
```

讨论

一些最重要的统计功能（例如，方差、标准差、偏差和峭度）都是以有关平均数的标准

www.TopSage.com

的样本动差术语定义的。这些统计功能的精确的定义在定义的文本上不同的地方还是有些差异的。关于这些统计功能，表 11-1 中的正文使用无偏见的定义。

表 11-1 统计功能的定义

统计功能	等式
n 次中心动差 (μ_n)	$\sum (x_i - \text{mean})^n P(x_i)$
方差	μ_2
标准差	$\sqrt{\mu_2}$
偏差	$\mu_3 / \mu_2^{3/2}$
峭度过剩	$(\mu_4 / \mu_2^2) - 3$

注意： 一个动差就是一序列数字的特性。换句话说，它是数学上描述一序列数字的一种方法。动差形成了几个重要的统计功能的基础，如方差、标准差、偏差和峭度。一个中心动差就是每一个原始数据与平均数之间计算出的动差。一个样本动差就是从一个离散的数字集而不是从一个函数上计算出来的动差。一个标准动差就是被标准方差的幂相除的动差（标准方差是第二个动差的平方根）。

编写一个统计函数的最简单的方法就是使用动差来定义它们。由于这里有几种不同的动差可用，每一种都接受一个常量型的整数值，所以我传入一个常量作为模板参数。这就能让编译器自动地产生更有效的代码，因为编译时就知道了这个整型数。

这个 moment 函数使用数学上的和操作符 $\sum$ 来定义。无论你什么时候想到和操作，你都应该想到来自 <numeric> 头文件中的 accumulate 函数。这个 accumulate 函数有两种形式：一种 accumulate 使用操作符 +，而另一种使用一个你提供的累加算符。你的累加算符接受两个值：到目前为止已累加的值和在这个序列中特定位置的值。

示例11-10通过说明这个由用户提供的算符在一个序列中的每一个元素上如何被反复调用说明了 accumulate 是如何工作的。

示例 11-10 Accumulate 函数的样本实现

```
template<class Iter_T, class Value_T, class BinOp_T>
Iter_T accumulate(Iter_T begin, Iter_T end, Value_T value, BinOp_T op) {
  while (begin != end) {
    value = op(value, *begin++)
  }
  return value;
}
```

11.6 生成随机数

问题

如何以一个统一的分布来在 [0.0, 1.0] 区间产生一个随机的浮点数？

www.TopSage.com

解决方案

在头文件<cstdlib>中，C++标准提供了一个C的运行函数rand来产生0到RAND_MAX之间的随机数。这个RAND_MAX宏代表这个rand函数可返回的最大值。示例11-11中说明如何使用rand函数来产生一个浮点数。

示例11-11 使用rand来产生随机数

```
#include <cstdlib>
#include <ctime>
#include <iostream>

using namespace std;

double doubleRand() {
  return double(rand()) / (double(RAND_MAX) + 1.0);
}

int main() {
  srand(static_cast<unsigned int>(clock()));
  cout << "expect 5 numbers within the interval [0.0, 1.0)" << endl;
  for (int i=0; i < 5; i++) {
    cout << doubleRand() << "\n";
  }
  cout << endl;
}
```

示例11-11中程序的输出如下：

```
expect 5 numbers within the interval [0.0, 1.0)
0.010437
0.740997
0.34906
0.369293
0.544373
```

讨论

更精确地说，随机数产生函数，包括rand函数，相对于真正的随机数，它返回事实上是一个假的随机数，因此无论我任何时候说随机数，事实上我都是指假随机数。

在使用这个rand函数之前，你需要调用srand来给这个随机数产生器产生一个种子(例如，初始化)。这就能保证随后的rand函数调用不会在每次程序运行时都产生相同的随机数序列。最简单地给这个随机数产生器种子的方法就是调用来自<ctime>头文件中的clock函数，以它的结果作为一个无符号整型数传给rand函数。重新给这个随机数产生器提供一个种子将导致产生的数字更不随机。

这个rand函数在很多方面都有限制。首先，它只产生整数，并且总是一个统一的分布来产生。第二，使用的特定随机数产生算符是依赖于实现的，因此，同样的种子在不同的平台上产生的随机数序列是不可能相同的。在某些种类的应用中，就可能出问题，如测试和调试时就有可能有问题。

你可以使用由Jens Maurer实现的Boost Random库代替这个rand函数，它使用TR1来产生随机数。

www.TopSage.com

注意：TR1 代表 Technical Report One，并且它使 C++ 98 标准类库的一个官方推荐扩展。

Boost Random库给整型数和浮点数类型都提供了更高质量的随机数产生函数，并且它还支持各种类型的分布。示例 11-12 说明了在[0,1]之间你能如何来产生随机的浮点数。

示例 11-12 使用 Boost 的 random 随机数类库

```
#include <boost/random.hpp>
#include <iostream>
#include <cstdlib>

using namespace std;
using namespace boost;

typedef boost::mt19937 BaseGenerator;
typedef boost::uniform_real<double> Distribution;
typedef boost::variate_generator<BaseGenerator, Distribution> Generator;

double boostDoubleRand() {
  static BaseGenerator base;
  static Distribution dist;
  static Generator rng(base, dist);
  return rng();
}

int main() {
  cout << "expect 5 numbers within the interval [0,1)" << endl;
  for (int i=0; i < 5; i++) {
    cout << boostDoubleRand() << "\n";
  }
  cout << endl;
}
```

Boost Random库的主要优点就是这个假的随机数产生算法是有保证的，并且基于选择的精确的算法产生的随机数是可重复的。示例 11-12 中，我使用了 Mersenne Twister 产生器（mt19937），因为它在性能和随机性上处理得很好。

11.7 使用随机数初始化容器

问题

如何使用一个随机数来填充一个任意的容器？

解决方案

你可以与一个返回随机数的算符一起使用来自<algorithm>头文件中的generate或者generate_n函数。请参考示例 11-13 中的例子。

示例 11-13 使用一个随机数来初始化容器

```
#include <algorithm>
#include <vector>
#include <iterator>
#include <iostream>
```

www.TopSage.com

```
#include <cstdlib>

using namespace std;

struct RndIntGen
{
  RndIntGen(int l, int h)
    : low(l), high(h)
  { }
  int operator()() const {
    return low + (rand() % ((high - low) + 1));
  }
private:
  int low;
  int high;
};

int main() {
  srand(static_cast<unsigned int>(clock()));
  vector<int> v(5);
  generate(v.begin(), v.end(), RndIntGen(1, 6));
  copy(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));
}
```

示例11-13中程序的输出如下：

```
3
1
2
6
4
```

讨论

标准C++的类库提供了函数generate和generate_n来使用一个产生器函数的结果填充容器。这些函数接受一个不带参数的算符（不带参数的函数指针或函数对象），它的结果将会赋给容器中连续的值。示例11-14中说明了这个generate和generate_n函数的一个实现例子。

示例11-14 generate和generate_n函数的实现例子

```
template<class Iter_T, class Fxn_T>
void generate(Iter_T first, Iter_T last, Fxn_T f) {
  while (first != last) *first++ = f();
}

template<class Iter_T, class Fxn_T>
void generate_n(Iter_T first, int n, Fxn_T f) {
  for (int i=0; i < n; ++i) *first++ = f();
}
```

11.8 表示一个数目可变的数字矢量

问题

如何实现一个类型，使之能够动态改变大小地操作数字矢量？

www.TopSage.com

大家网 TopSage.com

解决方案

你可以使用在<valarray>头文件中定义的valarray模板。示例11-15说明你该如何使用这个valarray模板。

示例11-15 使用valarray

```
#include <valarray>
#include <iostream>

using namespace std;

int main() {
  valarray<int> v(3);
  v[0] = 1; v[1] = 2; v[2] = 3;
  cout << v[0] << ", " << v[1] << ", " << v[2] << endl;
  v = v + v;
  cout << v[0] << ", " << v[1] << ", " << v[2] << endl;
  v /= 2;
  cout << v[0] << ", " << v[1] << ", " << v[2] << endl;
}
```

示例11-15中程序的输出如下：

```
1, 2, 3
2, 4, 6
1, 2, 3
```

讨论

不论它的名字如何，vector还是不希望作为一个数字矢量来使用的；但valarray则可以。由于valarray的设计，在C++的实现中，特别是在高性能的机器上，它可以应用特殊的矢量优化。valarray的另一个优点是它提供很多重载的操作符，特别是与数字矢量一起工作的。这些操作提供了这些功能如矢量相加和数量相乘等。

valarray模板也可以与一些标准算法如C风格的数组那样被使用。关于如何在一个valarray的开始处和结束处的后一项处创建相应的迭代器，请参考示例11-16。

示例11-16 得到这个valarray的迭代器

```
template<class T>
T* valarray_begin(valarray<T>& x) {
  return &x[0];
}

template<class T>
T* valarray_end(valarray<T>& x) {
  return valarray_begin(x) + x.size();
}
```

尽管看上去它有点学院派，你还是不应该通过写&x[x.size()]这样的代码来为一个valarray创建一个末端迭代器。即使它可以工作，那也只是偶然的，因为索引一个valarray的最后一个合法下标之后的项，其结果是不可预知的。

由于在valarray中缺乏begin和end成员函数，使得它只能是非STL类库。这种缺

乏强化了这个valarray并不是一个STL容器概念的模型。尽管这样，在需要一个随机访问的迭代器时，你还是可以把valarray用到任何泛型算法中。

11.9 表示一个数目固定的数字矢量

问题

如何实现一个有效的表示来操作一个固定大小的数字矢量？

解决方案

在很多平常的软件体系结构中，当这个矢量的大小在编译时可以确定时，使用一个自定义的矢量比使用一个可变数组valarray都更有效。示例11-17提供了关于一个固定大小的矢量模板的一个简单的例子实现，它叫做kvector。

示例11-17 kvector.hpp

```
#include <algorithm>
#include <cassert>

template<class Value_T, unsigned int N>
class kvector
{
public:
  // public fields
  Value_T m[N];

  // public typedefs
  typedef Value_T value_type;
  typedef Value_T* iterator;
  typedef const Value_T* const_iterator;
  typedef Value_T& reference;
  typedef const Value_T& const_reference;
  typedef size_t size_type;

  // shorthand for referring to kvector
  typedef kvector self;

  // member functions
  template<typename Iter_T>
  void copy(Iter_T first, Iter_T last) { copy(first, last, begin()); }
  iterator begin() { return m; }
  iterator end() { return m + N; }
  const_iterator begin() const { return m; }
  const_iterator end() const { return m + N; }
  reference operator[](size_type n) { return m[n]; }
  const_reference operator[](size_type n) const { return m[n]; }
  static size_type size() { return N; }

  // vector operations
  self& operator+=(const self& x) {
    for (int i=0; i<N; ++i) m[i] += x.m[i]; return *this;
  }
  self& operator-=(const self& x) {
    for (int i=0; i<N; ++i) m[i] -= x.m[i]; return *this;
  }

  // scalar operations
    self& operator=(value_type x) {
    std::fill(begin(  ), end(  ), x); return *this;
```

www.TopSage.com

```
  }
  self& operator+=(value_type x) {
    for (int i=0; i<N; ++i) m[i] += x; return *this;
  }
  self& operator-=(value_type x) {
    for (int i=0; i<N; ++i) m[i] -= x; return *this;
  }
  self& operator*=(value_type x) {
    for (int i=0; i<N;  ++i) m[i] *= x; return *this;
  }
  self& operator/=(value_type x) {
    for (int i=0; i<N; ++i) m[i] /= x; return *this;
  }
  self& operator%=(value_type x) {
    for (int i=0; i<N; ++i) m[i] %= x; return *this;
   }
  self operator-() {
    self x;
    for (int i=0; i<N; ++i) x.m[i] = -m[i];
    return x;
  }

  // friend operators
  friend self operator+(self x, const self& y) { return x += y; }
  friend self operator-(self x, const self& y) { return x -= y; }
  friend self operator+(self x, value_type y) { return x += y; }
  friend self operator-(self x, value_type y) { return x -= y; }
  friend self operator*(self x, value_type y) { return x *= y; }
  friend self operator/(self x, value_type y) { return x /= y; }
  friend self operator%(self x, value_type y) { return x %= y; }
};
```

示例 11-18 中说明了如何使用这个 kvector 类模板。

示例 11-18 使用 kvector

```
#include "kvector.hpp"

#include <algorithm>
#include <numeric>
#include <iostream>

using namespace std;

int main() {
  kvector<int, 4> v = { 1, 2, 3, 4 };
  cout << "sum = " << accumulate(v.begin(), v.end(), 0) << endl;
  v *= 3;
  cout << "sum = " << accumulate(v.begin(), v.end(), 0) << endl;
  v += 1;
  cout << "sum = " << accumulate(v.begin(), v.end(), 0) << endl;
}
```

示例 11-18 中程序的输出如下：

```
sum = 10
sum = 30
sum = 34
```

讨论

示例 11-17 中的 kvector 模板是一个可变数组 valarray 和一个 TR1 标准推荐的数组模板之间的交叉。就像可变数组 valarray 一样，kvector 代表了一个给定数字类型的一序列的值，但就像 tr1::array 一样，它的大小在编译时是确定的。

www.TopSage.com

这个kvector的特征是它提供了数组初始化语法并且提供了begin和end成员函数。在效果上，一个kvector可以认为是一个假的容器，这就是意味着它满足标准容器概念的一些而不是全部要求。这样就使得kvector比可变数组更容易与标准算法一起使用。

模板类kvector的另一个优点是它支持数组的初始化语法，如下所示：

```
int x;
kvector<int, 3> k = { x = 1, x + 2, 5 };
```

这个初始化语法是可能的，因为kvector是一个集合。一个集合是一个数组或一个类，并且这个类没有用户声明的构造函数，没有私有和保护类型的非静态数据成员，没有基类，并且没有虚函数。请注意，你仍可以以默认值来声明一个kvector，如下所示：

```
kvector<int, 3> k = {};
```

这样，这个矢量中的值都是0。

就像你能看到的那样，在全部满足标准容器要求和允许数组初始化语法之间，我不得不在设计上做一些折衷。同样的折衷也发生在TR1数组模板上。

大概，kvector比起动态矢量实现的最大优点是性能。这个kvector在两方面比最动态的矢量实现都更有效：编译器在固定大小循环上优化得更好，并且这里没有动态内存分配。对于小的矩阵（如，2 × 2或者3 × 3）性能上的区别就很明显，这些常常是发生在很多类型的应用中。

这个“self”的typedef的定义是什么意思呢？

示例11-17和随后的例子中我使用的“self”的typedef定义是一个方便的速记符，我使用它来指向当前类的类型。当使用“self”的typedef定义而不是使用这个类的全名时，它使得代码编写和理解都更容易。

11.10 点积计算

问题

有两个含有数字的容器，它们具有相同的长度，如何计算它们的点积？

解决方案

示例11-19说明了如何使用来自头文件<numeric>中的inner_product函数来计算点积。

示例11-19 计算点积

```
#include <numeric>
#include <iostream>
#include <vector>
```

www.TopSage.com

```
using namespace std;

int main() {
  int v1[] = { 1, 2, 3 };
  int v2[] = { 4, 6, 8 };
  cout << "the dot product of (1,2,3) and (4,6,8) is ";
  cout << inner_product(v1, v1 + 3, v2, 0) << endl;
}
```

示例 11-19 中程序的输出如下：

```
the dot product of (1,2,3) and (4,6,8) is 40
```

讨论

这个点积是内积的一种形式，叫做欧几里得内积。这个 inner_product 函数的声明如下：

```
template<class In, class In2, class T>
T inner_product(In first, In last, In2 first2, T init);
template<class In, class In2, class T, class BinOp, class BinOp2>
T inner_product(In first, In last, In2 first2, T init, BinOp op, Binop2 op2);
```

inner_product 函数的第一种形式是求来自这个两个容器中对应元素积之和。inner_product的第二种形式是让你能提供自己的成对操作和累积函数。请参考示例 11-20 关于说明这个 inner_product 函数是如何工作的一个例子。

示例 11-20 inner_product() 函数的一个例子实现

```
template<class In, class In2, class T, class BinOp, class BinOp2>
T inner_product(In first, In last, In2 first2, T init, BinOp op, Binop2 op2){
  while (first != last) {
    BinOp(init, BinOp2(*first++, *first2++));
  }
  return init;
}
```

因为它的灵活的实现，你可以为很多目的使用这个inner_product函数而不仅是计算点积（例如，你可以使用它来计算两个矢量之间的距离或者计算一个矢量的范数）。

请参考 11.11 节和 11.12 节。

11.11 计算矢量的范数

问题

如何计算一个数字矢量的范数（即长度）？

解决方案

你可以使用来自头文件<numeric>中的inner_product函数来使得一个矢量和它自己相乘，如示例 11-21 所示。

www.TopSage.com

大家网 TopSage.com

示例11-21 计算一个矢量的范数

```
#include <numeric>
#include <vector>
#include <cmath>
#include <iostream>

using namespace std;

template<typename Iter_T>
long double vectorNorm(Iter_T first, Iter_T last) {
  return sqrt(inner_product(first, last, first, 0.0L));
}

int main() {
  int v[] = { 3, 4 };
  cout << "The length of the vector (3,4) is ";
  cout << vectorNorm(v, v + 2) << endl;
}
```

示例11-21产生的输出如下：

```
The length of the vector (3,4) is 5
```

讨论

示例11-21使用来自头文件<numeric>中的inner_product函数来找到这个数字矢量和它自己的点积。这个点积的平方根就是这个矢量的范数或者长度。

在这个vectorNorm函数中，代替推演出这个结果类型，我选择返回一个long double类型，这样可以尽可能少地丢失精度，如果一个矢量是一序列的整数，那么它就可能不是一个实际的例子，这个长度也能够使用一个整型数来代替。

11.12 计算两个矢量之间的距离

问题

如何求解两个矢量之间的欧几里得距离？

解决方案

两个矢量之间的欧几里得距离定义为相应元素之间差的平方和的平方根。它可以如示例11-22中一样计算。

示例11-22 计算两个矢量之间的距离

```
#include <cmath>
#include <iostream>

using namespace std;

template<class Iter_T, class Iter2_T>
double vectorDistance(Iter_T first, Iter_T last, Iter2_T first2) {
  double ret = 0.0;
  while (first != last) {
```

www.TopSage.com

```
    double dist = (*first++) - (*first2++);
    ret += dist * dist;
  }
  return ret > 0.0 ? sqrt(ret) : 0.0;
}

int main() {
  int v1[] = { 1, 5 };
  int v2[] = { 4, 9 };
  cout << "distance between vectors (1,5) and (4,9) is ";
  cout << vectorDistance(v1, v1 + 2, v2) << endl;
}
```

示例11-22中程序的输出如下：

```
distance between vectors (1,5) and (4,9) is 5
```

讨论

示例11-22是一个直接的例子，它说明了如何按照STL的风格来写一个简单的泛型函数。为了计算矢量的距离，我可能代替它来使用这个inner_product函数，而不选择使用一个算符，因为它比严格需要的更复杂。示例11-23说明了你该如何使用一个算符和这个来自<numeric>头文件中的inner_product函数来计算矢量距离。

示例 11-23 使用 inner_product 来计算矢量之间的距离

```
#include <numeric>
#include <cmath>
#include <iostream>
#include <functional>

using namespace std;

template<class Value_T>
struct DiffSquared {
  Value_T operator()(Value_T x, Value_T y) const {
    return (x - y) * (x - y);
  }
};

template<class Iter_T, class Iter2_T>
double vectorDistance(Iter_T first, Iter_T last, Iter2_T first2)  {
  double ret = inner_product(first, last, first2, 0.0L,
    plus<double>(), DiffSquared<double>());
  return ret > 0.0 ? sqrt(ret) : 0.0;
}

int main() {
  int v1[] = { 1, 5 };
  int v2[] = { 4, 9 };
  cout << "distance between vectors (1,5) and (4,9) is ";
  cout << vectorDistance(v1, v1 + 2, v2) << endl;
}
```

因为inner_product()的一个实现可能包含对你的平台和编译器的特殊优化，所以当有可能时，我更愿意使用它。

www.TopSage.com

11.13 实现一个跨步迭代器

问题

有一个连续序列的数字，如何实现一次枚举跨过 n 个元素的迭代器？

解决方案

示例 11-24 中以一个独立的头文件说明了一个跨步迭代器。

示例 11-24 stride_iter.hpp

```
#ifndef STRIDE_ITER_HPP
#define STRIDE_ITER_HPP

#include <iterator>
#include <cassert>

template<class Iter_T>
class stride_iter
{
public:
  // public typedefs
  typedef typename std::iterator_traits<Iter_T>::value_type value_type;
  typedef typename std::iterator_traits<Iter_T>::reference reference;
  typedef typename std::iterator_traits<Iter_T>::difference_type
    difference_type;
  typedef typename std::iterator_traits<Iter_T>::pointer pointer;
  typedef std::random_access_iterator_tag iterator_category;
  typedef stride_iter self;

  // constructors
  stride_iter() : m(NULL), step(0) { };
  stride_iter(const self& x) : m(x.m), step(x.step) { }
  stride_iter(Iter_T x, difference_type n) : m(x), step(n) { }

  // operators
  self& operator++() { m += step; return *this; }
  self operator++(int) { self tmp = *this; m += step; return tmp; }
  self& operator+=(difference_type x) { m += x * step; return *this; }
  self& operator--() { m -= step; return *this; }
  self operator--(int) { self tmp = *this; m -= step; return tmp; }
  self& operator-=(difference_type x) { m -= x * step; return *this; }
  reference operator[](difference_type n) { return m[n * step]; }
  reference operator*() { return *m; }

  // friend operators
  friend bool operator==(const self& x, const self& y) {
    assert(x.step == y.step);
    return x.m == y.m;
  }
  friend bool operator!=(const self& x, const self& y) {
    assert(x.step == y.step);
    return x.m != y.m;
  }
  friend bool operator<(const self& x, const self& y) {
    assert(x.step == y.step);
    return x.m < y.m;
  }
  friend difference_type operator-(const self& x, const self& y) {
    assert(x.step == y.step);
    return (x.m - y.m) / x.step;
  }
```

www.TopSage.com

```
  friend self operator+(const self& x, difference_type y) {
    assert(x.step == y.step);
    return x += y * x.step;
  }
  friend self operator+(difference_type x, const self& y) {
    assert(x.step == y.step);
    return y += x * x.step;
  }
private:
  Iter_T m;
  difference_type step;
};

#endif
```

示例 11-25 说明了如何使用示例 11-24 中的 stride_iter 来一次枚举跨过两个元素的序列。

示例 11-25 使用 stride_iter

```
#include "stride_iter.hpp"

#include <algorithm>
#include <iterator>
#include <iostream>

using namespace std;

int main() {
  int a[] = { 0, 1, 2, 3, 4, 5, 6, 7 };
  stride_iter<int*> first(a, 2);
  stride_iter<int*> last(a + 8, 2);
  copy(first, last, ostream_iterator<int>(cout, "\n"));
}
```

示例 11-25 中程序的输出如下：

```
0
2
4
6
```

讨论

在矩阵的实现中，跨步迭代器是平常的。它们给实现作为一序列数字的矩阵提供了一个简单和有效的方法。示例11-24中实现的跨步迭代器是一个作为模板参数传给它的另外一个迭代器的包装迭代器。

我需要这个跨步迭代器与STL相兼容，因此我不得不选择标准迭代器概念中的一种并且满足它的要求。示例 11-24 中的跨步迭代器是一种随机访问迭代器的模式。

在示例11-26中，当这个步长在编译时知道时，我给跨步迭代器提供了一个独立的实现，叫做kstride_iter。由于这个步长是作为模板参数传递的，编译器就能为这个迭代器做更加有效的代码优化，并且能够减小这个迭代器的规模。

示例 11-26 kstride_iter.hpp

```
#ifndef KSTRIDE_ITER_HPP
#define KSTRIDE_ITER_HPP
```

www.TopSage.com

```
#include <iterator>

template<class Iter_T, int Step_N>
class kstride_iter
{
public:
  // public typedefs
  typedef typename std::iterator_traits<Iter_T>::value_type value_type;
  typedef typename std::iterator_traits<Iter_T>::reference reference;
  typedef typename
    std::iterator_traits<Iter_T>::difference_type difference_type;
  typedef typename std::iterator_traits<Iter_T>::pointer pointer;
  typedef std::random_access_iterator_tag iterator_category;
  typedef kstride_iter self;

  // constructors
  kstride_iter() : m(NULL) { }
  kstride_iter(const self& x) : m(x.m) { }
  explicit kstride_iter(Iter_T x) : m(x) { }

  // operators
  self& operator++() { m += Step_N; return *this; }
  self operator++(int) { self tmp = *this; m += Step_N; return tmp; }
  self& operator+=(difference_type x) { m += x * Step_N; return *this; }
  self& operator--() { m -= Step_N; return *this; }
  self operator--(int) { self tmp = *this; m -= Step_N; return tmp; }
  self& operator-=(difference_type x) { m -= x * Step_N; return *this; }
  reference operator[](difference_type n) { return m[n * Step_N]; }
  reference operator*() { return *m; }

  // friend operators
  friend bool operator==(self x, self y) { return x.m == y.m; }
  friend bool operator!=(self x, self y) { return x.m != y.m; }
  friend bool operator<(self x, self y) { return x.m < y.m; }
  friend difference_type operator-(self x, self y) {
    return (x.m - y.m) / Step_N;
  }
  friend self operator+(self x, difference_type y) { return x += y * Step_N; }
  friend self operator+(difference_type x, self y) { return y += x * Step_N; }
private:
  Iter_T m;
};

#endif
```

示例 11-27 说明了该如何使用这个 kstride_iter。

示例 11-27 使用 kstride_iter

```
#include "kstride_iter.hpp"

#include <algorithm>
#include <iterator>
#include <iostream>

using namespace std;

int main() {
  int a[] = { 0, 1, 2, 3, 4, 5, 6, 7 };
  kstride_iter<int*, 2> first(a);
  kstride_iter<int*, 2> last(a + 8);
  copy(first, last, ostream_iterator<int>(cout, "\n"));
}
```

www.TopSage.com

大家网 TopSage.com

11.14 实现一个大小可变的矩阵

问题

如何实现存储和表示一个数字矩阵？编译时该矩阵的维度（行数和列数）是未知的。

解决方案

使用来自11.12节中跨步迭代器和一个可变大小的数组valarray，示例11-28为一个可变大小的矩阵提供了一个总的目标和有效的实现。

示例 11-28 matrix.hpp

```
#ifndef MATRIX_HPP
#define MATRIX_HPP

#include "stride_iter.hpp" // see Recipe 11.12

#include <valarray>
#include <numeric>
#include <algorithm>

template<class Value_T>
class matrix
{
public:
  // public typedefs
  typedef Value_T value_type;
  typedef matrix self;
  typedef value_type* iterator;
  typedef const value_type* const_iterator;
  typedef Value_T* row_type;
  typedef stride_iter<value_type*> col_type;
  typedef const value_type* const_row_type;
  typedef stride_iter<const value_type*> const_col_type;

  // constructors
  matrix() : nrows(0), ncols(0), m() { }
  matrix(int r, int c) : nrows(r), ncols(c), m(r * c) { }
  matrix(const self& x) : m(x.m), nrows(x.nrows), ncols(x.ncols) { }

  template<typename T>
  explicit matrix(const valarray<T>& x)
  : m(x.size() + 1), nrows(x.size()), ncols(1)
  {
    for (int i=0; i<x.size(); ++i) m[i] = x[i];
  }

  // allow construction from matricies of other types
  template<typename T>
  explicit matrix(const matrix<T>& x)
  : m(x.size() + 1), nrows(x.nrows), ncols(x.ncols)
  {
    copy(x.begin(), x.end(), m.begin());
  }

  // public functions
  int rows() const { return nrows; }
  int cols() const { return ncols; }
  int size() const { return nrows * ncols; }

  // element access
  row_type row_begin(int n) { return &m[n * cols()]; }
```

www.TopSage.com

```
  row_type row_end(int n) { return row_begin() + cols(); }
  col_type col_begin(int n) { return col_type(&m[n], cols()); }
  col_type col_end(int n) { return col_begin(n) + cols(); }
  const_row_type row_begin(int n) const { return &m[n * cols()]; }
  const_row_type row_end(int n) const { return row_begin() + cols(); }
  const_col_type col_begin(int n) const { return col_type(&m[n], cols()); }
  const_col_type col_end(int n) const { return col_begin() + cols(); }
  iterator begin() { return &m[0]; }
  iterator end() { return begin() + size(); }
  const_iterator begin() const { return &m[0]; }
  const_iterator end() const { return begin() + size(); }

  // operators
  self& operator=(const self& x) {
    m = x.m; nrows = x.nrows; ncols = x.ncols; return *this;
  }
  self& operator=(value_type x) { m = x; return *this; }
  row_type operator[](int n) { return row_begin(n); }
  const_row_type operator[](int n) const { return row_begin(n); }
  self& operator+=(const self& x) { m += x.m; return *this; }
  self& operator-=(const self& x) { m -= x.m; return *this; }
  self& operator+=(value_type x) { m += x; return *this; }
  self& operator-=(value_type x) { m -= x; return *this; }
  self& operator*=(value_type x) { m *= x; return *this; }
  self& operator/=(value_type x) { m /= x; return *this; }
  self& operator%=(value_type x) { m %= x; return *this; }
  self operator-() { return -m; }
  self operator+() { return +m; }
  self operator!() { return !m; }
  self operator~() { return ~m; }

  // friend operators
  friend self operator+(const self& x, const self& y) { return self(x) += y; }
  friend self operator-(const self& x, const self& y) { return self(x) -= y; }
  friend self operator+(const self& x, value_type y) { return self(x) += y; }
  friend self operator-(const self& x, value_type y) { return self(x) -= y; }
  friend self operator*(const self& x, value_type y) { return self(x) *= y; }
  friend self operator/(const self& x, value_type y) { return self(x) /= y; }
  friend self operator%(const self& x, value_type y) { return self(x) %= y; }
private:
  mutable valarray<Value_T> m;
  int nrows;
  int ncols;
};

#endif
```

示例11-29说明该如何使用这个matrix模板类。

示例11-29 使用这个matrix模板

```
#include "matrix.hpp"

#include <iostream>

using namespace std;

int main() {
  matrix<int> m(2,2);
  m = 0;
  m[0][0] = 1;
  m[1][1] = 1;
  m *= 2;
  cout << "(" << m[0][0] << "," << m[0][1] << ")" << endl;
  cout << "(" << m[1][0] << "," << m[1][1] << ")" << endl;
}
```

www.TopSage.com

示例11-29中程序的输出如下：

```
(2,0)
(0,2)
```

讨论

示例11-28中这个matrix模板的设计灵感来自《C++编程语言》第三版（Addison Wesley）中的Bjarne Stroustrup的矩阵模板。Stroustrup的实现区别于它就在于为索引使用了片断和指向一个大小可变数组valarray的指针。示例11-17中的matrix实现使用的是来自11.12节的跨步迭代器，这样就使得这个迭代器更加紧凑并且在实现上更加有效。

这个matrix模板类让一个元素的索引可以通过使用两个下标操作指定行数和列数来进行索引。如下所示：

```
matrix<int> m(100,100);
cout << "the element at row 24 and column 42 is " << m[24][42] << endl;
```

并且这个matrix模板类也提供了begin和end成员函数，这就意味着它也能很容易地与各种各样的STL算法一起使用。

示例11-28中的一行代码或许可能令你费解。这行代码就是这个声明：

```
mutable valarray<Value_T> m;
```

这个成员字段m使用mutable来声明肯定会带来麻烦。如果它不是因为这行代码，我将不可能提供一个常量型的迭代器，因为你不可能在一个常量型的valarray上创建一个迭代器。

请参考11.15节和11.16节。

11.15 实现一个大小固定的矩阵

问题

如何实现一个有效的矩阵实现？该矩阵的维度（例如，行数和列数）在编译时都是常数。

解决方案

当一个矩阵的维度在编译时是常数时，编译器就可以很容易地对它进行优化，如在示例11-30中，矩阵的行数和列数是作为模板参数传递的。

示例11-30 kmatrix.hpp

```
#ifndef KMATRIX_HPP
#define KMATRIX_HPP

#include "kvector.hpp"
#include "kstride_iter.hpp"
```

www.TopSage.com

```
template<class Value_T, int Rows_N, int Cols_N>
class kmatrix
{
public:
  // public typedefs
  typedef Value_T value_type;
  typedef kmatrix self;
  typedef Value_T* iterator;
  typedef const Value_T* const_iterator;
  typedef kstride_iter<Value_T*, 1> row_type;
  typedef kstride_iter<Value_T*, Cols_N> col_type;
  typedef kstride_iter<const Value_T*, 1> const_row_type;
  typedef kstride_iter<const Value_T*, Cols_N> const_col_type;

  // public constants
  static const int nRows = Rows_N;
  static const int nCols = Cols_N;

  // constructors
  kmatrix() { m = Value_T(); }
  kmatrix(const self& x) { m = x.m; }
  explicit kmatrix(Value_T& x) { m = x.m; }

  // public functions
  static int rows() { return Rows_N; }
  static int cols() { return Cols_N; }
  row_type row(int n) { return row_type(begin() + (n * Cols_N)); }
  col_type col(int n) { return col_type(begin() + n); }
  const_row_type row(int n) const {
    return const_row_type(begin() + (n * Cols_N));
  }
  const_col_type col(int n) const {
    return const_col_type(begin() + n);
  }
  iterator begin() { return m.begin(); }
  iterator end() { return m.begin() + size(); }
  const_iterator begin() const { return m; }
  const_iterator end() const { return m + size(); }
  static int size() { return Rows_N * Cols_N; }

  // operators
  row_type operator[](int n) { return row(n); }
  const_row_type operator[](int n) const { return row(n); }

  // assignment operations
  self& operator=(const self& x) { m = x.m; return *this; }
  self& operator=(value_type x) { m = x; return *this; }
  self& operator+=(const self& x) { m += x.m; return *this; }
  self& operator-=(const self& x) { m -= x.m; return *this; }
  self& operator+=(value_type x) { m += x; return *this; }
  self& operator-=(value_type x) { m -= x; return *this; }
  self& operator*=(value_type x) { m *= x; return *this; }
  self& operator/=(value_type x) { m /= x; return *this; }
  self operator-() { return self(-m); }

  // friends
  friend self operator+(self x, const self& y) { return x += y; }
  friend self operator-(self x, const self& y) { return x -= y; }
  friend self operator+(self x, value_type y) { return x += y; }
  friend self operator-(self x, value_type y) { return x -= y; }
  friend self operator*(self x, value_type y) { return x *= y; }
  friend self operator/(self x, value_type y) { return x /= y; }
  friend bool operator==(const self& x, const self& y) { return x != y; }
  friend bool operator!=(const self& x, const self& y) { return x.m != y.m; }
private:
  kvector<Value_T, (Rows_N + 1) * Cols_N> m;
};

#endif
```

www.TopSage.com

示例11-31是一个说明该如何使用这个kmatrix模板类的小程序。

示例11-31 使用kmatrix

```
#include "kmatrix.hpp"

#include <iostream>

using namespace std;

template<class Iter_T>
void outputRowOrColumn(Iter_T iter, int n) {
  for (int i=0; i < n; ++i) {
    cout << iter[i] << " ";
  }
  cout << endl;
}

template<class Matrix_T>
void initializeMatrix(Matrix_T& m) {
  int k = 0;
  for (int i=0; i < m.rows(); ++i) {
    for (int j=0; j < m.cols(); ++j) {
      m[i][j] = k++;
    }
  }
}

template<class Matrix_T>
void outputMatrix(Matrix_T& m) {
  for (int i=0; i < m.rows(); ++i) {
    cout << "Row " << i << " = ";
    outputRowOrColumn(m.row(i), m.cols());
  }
  for (int i=0; i < m.cols(); ++i) {
    cout << "Column " << i << " = ";
    outputRowOrColumn(m.col(i), m.rows());
  }
}

int main()
{
  kmatrix<int, 2, 4> m;
  initializeMatrix(m);
  m *= 2;
  outputMatrix(m);
}
```

示例11-31中程序的输出如下：

```
Row 0 = 0 2 4 6
Row 1 = 8 10 12 14
Column 0 = 0 8
Column 1 = 2 10
Column 2 = 4 12
Column 3 = 6 14
```

讨论

示例11-30和示例11-31中kmatrix类模板的设计和使用与11.14节中的matrix类模板十分相似。唯一的重大区别就是当声明一个kmatrix实例时，你是作为模板参数传递维度的，如下所示：

```
kmatrix<int, 5, 6> m; // declares a matrix with five rows and six columns
```

www.TopSage.com

对于任何需要矩阵的应用程序来说，在编译时就知晓它的维度是一个共同的要求。作为模板参数传递行数和列数使得编译器能够更容易地来使用一些通常的优化，如解开循环、函数内嵌和快速索引等。

注意：就像前面讨论的常量长度的矢量(kvector)一样，这个kmatrix模板当使用小规模的矩阵时特别有效。

请参考11.14节和11.16节。

11.16 矩阵相乘

问题

如何对两个矩阵进行高效的乘法运算？

解决方案

示例11-32说明了矩阵乘法的一种实现，它可应用于动态大小或固定大小矩阵的实现。这个算法技术产生的结果与等式A=A+B*C相同的结果，或许你会觉得吃惊，它比A=B*C等式的效率还高。

示例11-32 矩阵相乘

```
#include "matrix.hpp" // recipe 11.13
#include "kmatrix.hpp" // recipe 11.14
#include <iostream>
#include <cassert>

using namespace std;

template<class M1, class M2, class M3>
void matrixMultiply(const M1& m1, const M2& m2, M3& m3)
{
  assert(m1.cols() == m2.rows());
  assert(m1.rows() == m3.rows());
  assert(m2.cols() == m3.cols());
  for (int i=m1.rows()-1; i >= 0; --i) {
    for (int j=m2.cols()-1; j >= 0; --j) {
      for (int k = m1.cols()-1; k >= 0; --k) {
        m3[i][j] += m1[i][k] * m2[k][j];
      }
    }
  }
}

int main()
{
  matrix<int> m1(2, 1);
  matrix<int> m2(1, 2);
  kmatrix<int, 2, 2> m3;
  m3 = 0;
  m1[0][0] = 1; m1[1][0] = 2;
  m2[0][0] = 3; m2[0][1] = 4;
  matrixMultiply(m1, m2, m3);
```

www.TopSage.com

```
  cout << "(" << m3[0][0] << ", " << m3[0][1] << ")" << endl;
  cout << "(" << m3[1][0] << ", " << m3[1][1] << ")" << endl;
}
```

示例 11-32 的输出如下：

```
(3, 4)
(6, 8)
```

讨论

当两个矩阵相乘时，第一个矩阵的列数必须与第二个矩阵的行数相同。结果矩阵中的行数与第一个矩阵的行数相同，列数与第二个矩阵的列数相同。我使用了头文件<cassert>中的assert宏来确保调试版本中的这个条件为真。

高效的矩阵相乘的关键是避免过多的临时变量的创建和复制。因此，示例11-32中矩阵乘法函数使用引用来传递结果。如果通过改写操作符*来这样直接地实现这个乘法操作，它将会导致一个临时矩阵的分配、复制和释放操作。当处理大矩阵时，这将会导致潜在的性能问题。

注意： 示例11-32中计算A=A+B*C矩阵等式而不是A=B*C等式是为了避免不必要的对矩阵A的初始化。

请参考 11.17 节。

11.17 计算快速傅里叶转换

问题

如何使用快速傅里叶转换（FFT）算法来高效地计算离散傅里叶转换（DFT）？

解决方案

示例 11-33 中的代码提供了 FFT 的一个基本实现。

示例 11-33 FFT implementation FFT 实现

```
#include <iostream>
#include <complex>
#include <cmath>
#include <iterator>

using namespace std;

unsigned int bitReverse(unsigned int x, int log2n) {
  int n = 0;
  int mask = 0x1;
  for (int i=0; i < log2n; i++) {
    n <<= 1;
    n |= (x & 1);
    x >>= 1;
```

www.TopSage.com

```
  }
  return n;
}

const double PI = 3.1415926536;

template<class Iter_T>
void fft(Iter_T a, Iter_T b, int log2n)
{
  typedef typename iterator_traits<Iter_T>::value_type complex;
  const complex J(0, 1);
  int n = 1 << log2n;
  for (unsigned int i=0; i < n; ++i) {
    b[bitReverse(i, log2n)] = a[i];
  }
  for (int s = 1; s <= log2n; ++s) {
    int m = 1 << s;
    int m2 = m >> 1;
    complex w(1, 0);
    complex wm = exp(-J * (PI / m2));
    for (int j=0; j < m2; ++j) {
      for (int k=j; k < n; k += m) {
        complex t = w * b[k + m2];
        complex u = b[k];
        b[k] = u + t;
        b[k + m2] = u - t;
      }
      w *= wm;
    }
  }
}

int main() {
  typedef complex<double> cx;
  cx a[] = { cx(0,0), cx(1,1), cx(3,3), cx(4,4),
    cx(4, 4), cx(3, 3), cx(1,1), cx(0,0) };
  cx b[8];
  fft(a, b, 3);
  for (int i=0; i<8; ++i)
    cout << b[i] << "\n";
}
```

示例11-33中程序的输出如下：

```
(16,16)
(-4.82843,-11.6569)
(0,0)
(-0.343146,0.828427)
(0,0)
(0.828427,-0.343146)
(0,0)
(-11.6569,-4.82843)
```

讨论

对于频谱的分析，傅里叶转换是一个重要的等式，并且在很多工程和科学应用中经常需要用到。FFT是一个计算DFT的算法，其复杂度是Nlog2(N)而不是DFT的本身实现的N^2的复杂度。FFT通过去掉一些冗余的计算从而提高了速度。

找到一个使用惯用C++实现的好的FFT实现（如，这些C++实现并不是从老的Fortran或C算法移植过来的）并且这样就不会受到版权许可方面的严格限制。示例11-3是基于

www.TopSage.com

公共领域方面的代码，它可以在usenet (*comp.dsp*) 上的数字信号处理新闻组中找到。比起更为通常的C风格的FFT实现，一个惯用的C++方案的优点是标准类库提供了complex模板，它减少了代码的数量。示例11-33中的fft()函数尽量写得简单而不时主要关注在性能上。

11.18 使用极坐标

问题

如何表示和操作极坐标?

解决方案

<complex>头文件中的complex模板提供了各种函数来转换极坐标。示例11-34说明了如何使用这个complex模板类来表示和操作极坐标。

示例11-34 使用complex模板类来表示极坐标

```
#include <complex>
#include <iostream>

using namespace std;

int main()  {
  double rho = 3.0; // magnitude
  double theta = 3.141592 / 2; // angle
  complex<double> coord = polar(rho, theta);
  cout << "rho = " << abs(coord) << ", theta = " << arg(coord) << endl;
  coord += polar(4.0, 0.0);
  cout << "rho = " << abs(coord) << ", theta = " << arg(coord) << endl;
}
```

示例11-34产生的输出如下：

```
rho = 3, theta = 1.5708
rho = 5, theta = 0.643501
```

讨论

在极坐标和复数之间自然地存在关系。尽管它们在一定程度上是可以互换的，但总的来说使用同样的类型代表不同的概念不是一个好主意。既然使用complex模板来表示极坐标是不雅的，示例11-35中我提供了一个极坐标类，它的使用则更为自然。

示例11-35 一个极坐标类

```
#include <complex>
#include <iostream>

using namespace std;

template<class T>
struct BasicPolar
{
  public:
    typedef BasicPolar self;
```

www.TopSage.com

```
    // constructors
    BasicPolar() : m() {  }
    BasicPolar(const self& x) : m(x.m) { }
    BasicPolar(const T& rho, const T& theta) : m(polar(rho, theta)) { }

    // assignment operations
    self operator-() { return Polar(-m); }
    self& operator+=(const self& x) { m += x.m; return *this; }
    self& operator-=(const self& x) { m -= x.m; return *this; }
    self& operator*=(const self& x) { m *= x.m; return *this; }
    self& operator/=(const self& x) { m /= x.m; return *this; }
    operator complex<T>() const { return m; }

    // public member functions
    T rho() const { return abs(m); }
    T theta() const { return arg(m); }

    // binary operations
    friend self operator+(self x, const self& y) { return x += y; }
    friend self operator-(self x, const self& y) { return x -= y; }
    friend self operator*(self x, const self& y) { return x *= y; }
    friend self operator/(self x, const self& y) { return x /= y; }

    // comparison operators
    friend bool operator==(const self& x, const self& y) { return x.m == y.m; }
    friend bool operator!=(const self& x, const self& y) { return x.m != y.m; }
  private:
    complex<T> m;
};

typedef BasicPolar<double> Polar;

int main()  {
  double rho = 3.0; // magnitude
  double theta = 3.141592 / 2; // angle
  Polar coord(rho, theta);
  cout << "rho = " << coord.rho() << ", theta = " << coord.theta() << endl;
  coord += Polar(4.0, 0.0);
  cout << "rho = " << coord.rho() << ", theta = " << coord.theta() << endl;
  system("pause");
}
```

在示例11-35中，我定义了一个Polar作为BasicPolar模板的typedef特殊化。这样你就有一种默认的方便类型，但如果你愿意，你仍可以使用另一种数字类型来特殊化这个BasicPolar模板。在标准类库中也用到了这项技术，string类就是basic_string模板的特殊化。

11.19 在位集上进行算术运算

问题

如何在一个位集上执行一些数学和比较操作，就好像它是一个代表无符号整型数一样的二进制数？

解决方案

示例11-36中的函数提供了这些能够在一个bitset类模板上执行数学和比较操作的函数，就好像这个位集代表一个无符号整型数一样，这个bitset类模板来自<bitset>头文件。

www.TopSage.com

大家网 TopSage.com

示例 11-36 bitset_arithmetic.hpp 文件

```
#include <stdexcept>
#include <bitset>

bool fullAdder(bool b1, bool b2, bool& carry) {
  bool sum = (b1 ^ b2) ^ carry;
  carry = (b1 && b2) || (b1 && carry) || (b2 && carry);
  return sum;
}

bool fullSubtractor(bool b1, bool b2, bool& borrow) {
  bool diff;
  if (borrow) {
    diff = !(b1 ^ b2);
    borrow = !b1 || (b1 && b2);
  }
  else {
    diff = b1 ^ b2;
    borrow = !b1 && b2;
  }
  return diff;
}

template<unsigned int N>
bool bitsetLtEq(const std::bitset<N>& x, const std::bitset<N>& y)
{
  for (int i=N-1; i >= 0; i--) {
    if (x[i] && !y[i]) return false;
    if (!x[i] && y[i]) return true;
  }
  return true;
}

template<unsigned int N>
bool bitsetLt(const std::bitset<N>& x, const std::bitset<N>& y)
{
  for (int i=N-1; i >= 0; i--) {
    if (x[i] && !y[i]) return false;
    if (!x[i] && y[i]) return true;
  }
  return false;
}

template<unsigned int N>
bool bitsetGtEq(const std::bitset<N>& x, const std::bitset<N>& y)
{
  for (int i=N-1; i >= 0; i--) {
    if (x[i] && !y[i]) return true;
    if (!x[i] && y[i]) return false;
  }
  return true;
}

template<unsigned int N>
bool bitsetGt(const std::bitset<N>& x, const std::bitset<N>& y)
{
  for (int i=N-1; i >= 0; i--) {
    if (x[i] && !y[i]) return true;
    if (!x[i] && y[i]) return false;
  }
  return false;
}

template<unsigned int N>
void bitsetAdd(std::bitset<N>& x, const std::bitset<N>& y)
{
```

www.TopSage.com

```
  bool carry = false;
  for (int i = 0; i < N; i++) {
    x[i] = fullAdder(x[i], y[i], carry);
  }
}

template<unsigned int N>
void bitsetSubtract(std::bitset<N>& x, const std::bitset<N>& y) {
  bool borrow = false;
  for (int i = 0; i < N; i++) {
    if (borrow) {
     if (x[i]) {
       x[i] = y[i];
       borrow = y[i];
 }
 else {
   x[i] = !y[i];
   borrow = true;
 }
    }
    else {
      if (x[i]) {
        x[i] = !y[i];
        borrow = false;
      }
      else {
        x[i] = y[i];
        borrow = y[i];
      }
    }
  }
}

template<unsigned int N>
void bitsetMultiply(std::bitset<N>& x, const std::bitset<N>& y)
{
  std::bitset<N> tmp = x;
  x.reset();

  // we want to minimize the number of times we shift and add
  if (tmp.count() < y.count()) {
    for (int i=0; i < N; i++)
      if (tmp[i]) bitsetAdd(x, y << i);
  }
  else {
    for (int i=0; i < N; i++)
      if (y[i]) bitsetAdd(x, tmp << i);
  }
}

template<unsigned int N>
void bitsetDivide(std::bitset<N> x, std::bitset<N> y,
  std::bitset<N>& q, std::bitset<N>& r)
{
  if (y.none()) {
    throw std::domain_error("division by zero undefined");
  }
  q.reset();
  r.reset();
  if (x.none()) {
    return;
  }
  if (x == y) {
    q[0] = 1;
    return;
  }
  r = x;
  if (bitsetLt(x, y)) {
```

www.TopSage.com

```
    return;
  }

  // count significant digits in divisor and dividend
  unsigned int sig_x;
  for (int i=N-1; i>=0; i--) {
    sig_x = i;
    if (x[i]) break;
  }
  unsigned int sig_y;
  for (int i=N-1; i>=0; i--) {
    sig_y = i;
    if (y[i]) break;
  }

  // align the divisor with the dividend
  unsigned int n = (sig_x - sig_y);
  y <<= n;

  // make sure the loop executes the right number of times
  n += 1;

  // long division algorithm, shift, and subtract
  while (n--)
  {
    // shift the quotient to the left
    if (bitsetLtEq(y, r))
    {
     // add a new digit to quotient
     q[n] = true;
     bitsetSubtract(r, y);
    }
    // shift the divisor to the right
    y >>= 1;
  }
}
```

示例 11-37 说明了你该如何使用这个 *bitset_arithmetic.hpp* 头文件。

示例 11-37 使用 bitset_arithmetic.hpp 中的函数

```
#include "bitset_arithmetic.hpp"

#include <bitset>
#include <iostream>
#include <string>

using namespace std;

int main() {
  bitset<10> bits1(string("100010001"));
  bitset<10> bits2(string("000000011"));
  bitsetAdd(bits1, bits2);
  cout << bits1.to_string<char, char_traits<char>, allocator<char> >() << endl;
}
```

示例 11-37 中程序的输出如下：

```
0100010100
```

讨论

这个bitset类模板给操作位集提供了一些基本的操作，但没有提供任何数学和比较操作。这是因为这个库不能安全地假设程序员可能期待任何位集所代表的数字类型的种类。

www.TopSage.com

示例11-36中的函数把一个bitset当成是一个无符号整型类型，并且为加、减、乘、除和比较提供了函数。这些函数为实现用户指定大小的整型类型提供了基础，并且11.20节就是这样用的。

在示例11-36中，我并没有使用最有效的算法。相反我选择了可能是最简单的算法，因为它们最容易理解。一个更加有效的算法实现与之相似，但应该操作在字上而不是单个位上。

请参考书11.20节。

11.20 表示固定宽度的大整数

问题

如何对一个比长整型所能表示的整数还要大的数执行一些算术操作？

解决方案

示例11-38中的BigInt模板使用了<bitset>头文件中的bitset，使得你可以使用一个固定位数作为模板参数来代表一个无符号整数。

示例11-38 big_int.hpp

```
#ifndef BIG_INT_HPP
#define BIG_INT_HPP

#include <bitset>

#include "bitset_arithmetic.hpp" // Recipe 11.20

template<unsigned int N>
class BigInt
{
  typedef BigInt self;
public:
  BigInt() : bits() { }
  BigInt(const self& x) : bits(x.bits) { }
  BigInt(unsigned long x) {
    int n = 0;
    while (x) {
      bits[n++] = x & 0x1;
      x >>= 1;
    }
  }
  explicit BigInt(const std::bitset<N>& x) : bits(x) { }

  // public functions
  bool operator[](int n) const { return bits[n]; }
  unsigned long toUlong() const { return bits.to_ulong( ); }

  // operators
  self& operator<<=(unsigned int n) {
    bits <<= n;
    return *this;
  }
  self& operator>>=(unsigned int n) {
    bits >>= n;
```

www.TopSage.com

```
    return *this;
  }
  self operator++(int) {
    self i = *this;
    operator++();
    return i;
  }
  self operator--(int) {
    self i = *this;
    operator--();
    return i;
  }
  self& operator++() {
    bool carry = false;
    bits[0] = fullAdder(bits[0], 1, carry);
    for (int i = 1; i < N; i++) {
bits[i] = fullAdder(bits[i], 0, carry);
    }
    return *this;
  }
  self& operator--() {
    bool borrow = false;
    bits[0] = fullSubtractor(bits[0], 1, borrow);
    for (int i = 1; i < N; i++) {
bits[i] = fullSubtractor(bits[i], 0, borrow);
    }
    return *this;
  }
  self& operator+=(const self& x) {
    bitsetAdd(bits, x.bits);
    return *this;
  }
  self& operator-=(const self& x) {
    bitsetSubtract(bits, x.bits);
    return *this;
  }
  self& operator*=(const self& x) {
    bitsetMultiply(bits, x.bits);
    return *this;
  }
  self& operator/=(const self& x) {
    std::bitset<N> tmp;
    bitsetDivide(bits, x.bits, bits, tmp);
    return *this;
  }
  self& operator%=(const self& x) {
    std::bitset<N> tmp;
    bitsetDivide(bits, x.bits, tmp, bits);
    return *this;
  }
  self operator~() const { return ~bits; }
  self& operator&=(self x) { bits &= x.bits; return *this; }
  self& operator|=(self x) { bits |= x.bits; return *this; }
  self& operator^=(self x) { bits ^= x.bits; return *this; }

  // friend functions
  friend self operator<<(self x, unsigned int n) { return x <<= n; }
  friend self operator>>(self x, unsigned int n) { return x >>= n; }
  friend self operator+(self x, const self& y) { return x += y; }
  friend self operator-(self x, const self& y) { return x -= y; }
  friend self operator*(self x, const self& y) { return x *= y; }
  friend self operator/(self x, const self& y) { return x /= y; }
  friend self operator%(self x, const self& y) { return x %= y; }
  friend self operator^(self x, const self& y) { return x ^= y; }
  friend self operator&(self x, const self& y) { return x &= y; }
  friend self operator|(self x, const self& y) { return x |= y; }
```

www.TopSage.com

大家网 TopSage.com

```
  // comparison operators
  friend bool operator==(const self& x, const self& y) {
    return x.bits == y.bits;
  }
  friend bool operator!=(const self& x, const self& y) {
    return x.bits != y.bits;
  }
  friend bool operator>(const self& x, const self& y) {
    return bitsetGt(x.bits, y.bits);
  }
  friend bool operator<(const self& x, const self& y) {
    return bitsetLt(x.bits, y.bits);
  }
  friend bool operator>=(const self& x, const self& y) {
    return bitsetGtEq(x.bits, y.bits);
  }
  friend bool operator<=(const self& x, const self& y) {
    return bitsetLtEq(x.bits, y.bits);
  }
private:
  std::bitset<N> bits;
};

#endif
```

这个 BigInt 模板类可以被使用来代表阶乘数，如示例 11-39 所示。

示例 11-39 使用这个 big_int 类

```
#include "big_int.hpp"

#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>

using namespace std;

void outputBigInt(BigInt<1024> x) {
  vector<int> v;
  if (x == 0) {
    cout << 0;
    return;
  }
  while (x > 0) {
    v.push_back((x % 10).to_ulong());
    x /= 10;
  }
  copy(v.rbegin(), v.rend(), ostream_iterator<int>(cout, ""));
  cout << endl;
}

int main() {
  BigInt<1024> n(1);
  // compute 32 factorial
  for (int i=1; i <= 32; ++i) {
    n *= i;
  }
  outputBigInt(n);
}
```

示例 11-39 中程序的输出如下：

```
263130836933693530167218012160000000
```

讨论

在很多应用中大整数是很平常的。如在密码学中，1000位的整型数或者是更大的数都是很平常的。然而，目前的C++标准只支持long int类型范围的整数。

注意： 长整型中的位数实现得必将特殊，但保证最少有32位，但是它不可能比1000大。请记住这些位中有一位是保留作符号位的。

标准的下一个版本（C++ 0x）希望能遵循C99标准，并且能够提供一个long long类型，这个类型将定义成最少和long int一样大，并且可能更大。尽管这样，还是有可能需要一个比最大的原始类型大的整型类型。

上面我说明的实现是基于使用bitset所代表的二进制位数，当然在性能上还是有些影响的。虽然如此，但带来的是简单易懂。一个任意精度的更有效的实现在这个例子的基础上也很容易实现。

请参考11.19节。

11.21 实现固定小数位的数值

问题

如何使用固定小数位而不是浮动小数位类型来代表一个实数，从而执行一些计算操作？

解决方案

示例11-40提供了一个固定小数位实数的实现，这里二进制小数点右边位置上的数字是一个模板参数。例如，basic_fixed_real<10>在二进制小数点右边有10个二进制数字，它使得代表数字的精度达到1/1 024。

示例11-40 使用一个固定小数实现来代表实数

```
#include <iostream>

using namespace std;

template<int E>
struct BasicFixedReal
{
  typedef BasicFixedReal self;
  static const int factor = 1 << (E - 1);
  BasicFixedReal() : m(0) { }
  BasicFixedReal(double d) : m(static_cast<int>(d * factor)) { }
  self& operator+=(const self& x) { m += x.m; return *this; }
  self& operator-=(const self& x) { m -= x.m; return *this; }
  self& operator*=(const self& x) { m *= x.m; m >>= E; return *this; }
  self& operator/=(const self& x) { m /= x.m; m *= factor; return *this; }
  self& operator*=(int x) { m *= x; return *this; }
  self& operator/=(int x) { m /= x; return *this; }
```

www.TopSage.com

```
  self operator-() { return self(-m); }
  double toDouble() const { return double(m) / factor;  }

  // friend functions
  friend self operator+(self x, const self& y) { return x += y; }
  friend self operator-(self x, const self& y) { return x -= y; }
  friend self operator*(self x, const self& y) { return x *= y; }
  friend self operator/(self x, const self& y) { return x /= y; }

  // comparison operators
  friend bool operator==(const self& x, const self& y) { return x.m == y.m; }
  friend bool operator!=(const self& x, const self& y) { return x.m != y.m; }
  friend bool operator>(const self& x, const self& y) { return x.m > y.m; }
  friend bool operator<(const self& x, const self& y) { return x.m < y.m; }
  friend bool operator>=(const self& x, const self& y) { return x.m >= y.m; }
  friend bool operator<=(const self& x, const self& y) { return x.m <= y.m; }
private:
  int m;
};

typedef BasicFixedReal<10> FixedReal;

int main() {
  FixedReal x(0);
  for (int i=0; i < 100; ++i) {
    x += FixedReal(0.0625);
  }
  cout << x.toDouble() << endl;
}
```

示例11-40中程序的输出如下：

```
    6.25
```

讨论

一个固定小数位数字就像一个浮动小数位数字一样，都是一个合适的代表一个实数的方式。一个浮动小数位数字存储的形式是：一个假数（m）和一个幂（e）形成一个等式 $m * b^e$，其中 b 是一个常数。

一个固定小数位数字基本上相同，但这个幂e也是一个常数。这个常数是在示例11-40中作为模板参数传给 `basic_fixed_real` 的。

通过把e作为一个常数来对待，它就能够让固定小数位数字在内部作为整型数来对待，并且对于在它们上面的数学操作就可以使用整型数的数学操作来执行。常常这样就可以提高基本数学运算操作的速度，特别是加法和减法。

固定小数位类型没有浮动小数位类型那样灵活，因为它们仅能代表一个窄范围的值。示例11-40中类型 `fixed_real` 代表的值的范围是从 -2 097 151～+2 097 151，且精度是1/1 024。

在固定小数位数字上实现加法和减法是非常直接的：我只用在这个潜在的数字上执行加和减就可以。为了执行除法和乘法操作，我就需要一些额外的为这个二进制小数点在这个假数上执行左移或右移操作。

www.TopSage.com

第12章

多线程

12.0 概述

本章描述了如何使用由William Kempf实现的Boost线程库用C++语言来实现多线程程序。Boost是一个开源可移植的高性能类库集合，它包含从简单的数据结构到复杂的分析框架。Boost线程库是一个多线程的框架。更多的关于Boost的信息，请参考*www.boost.org*。

标准C++本身并不支持多线程，因此你不能像使用其他的一些标准库的类如`string`、`vector`、`list`等那样来写出可移植的多线程代码。而Boost线程库就是在为实现一个标准的可移植的多线程上做的一些努力，并且它旨在使得那些多线程的共同的一些令人头痛的问题降到最少。

当你使用其他的标准库或者第三方类库时，只需把它解压缩到一个目录下面，然后使用`#include`宏把它包含到你的头文件中，然后就使用它来编码，然而使用多线程库可不这么简单。即使是一个很小的多线程应用程序，你都必须小心使用一些已经证明过的设计模式和一些公认的技巧来避免错误的出现。典型地，在一个单线程的应用系统中，你很容易发现一些通常的编程错误：off-by-one循环、引用一个空的或已删除的指针、浮点数转换过程中精度的丧失等等。而多线程程序就很不相同。不但是在你调试器中跟踪多个线程的运行很乏味，而且多线程程序是非确定的，也就是说程序的错误可能仅在一个很罕见的或非常复杂的环境中才出现。

这一章不是给你介绍关于多线程编程的。如果你以前使用过线程来写程序，但没有使用C++或者这个Boost线程库，那么这一章非常适合你。但这一章并不介绍那些基本的多线程编程，关于这些你需要读一些其他的有关多线程的书籍，尽管这些主题的著作比较少见，因为大部分程序员都不使用线程（尽管他们不应该这样）。

www.TopSage.com

大部分的Boost库文档和接下来的章节都讨论那些实用概念/模型的类。一个概念就是某个事物的一个抽象的描述，常常是一个类和它的行为，但不考虑它的实现。典型地，这个描述包含构造行为和析构行为，并且每一个这种方法都包含它们的前置条件、参数和后置条件。例如，互斥体的概念就是一个能被某个线程在某个时候锁定或解锁的事物。一个模型就是一个概念的一个具体表现，如Boost线程库中的mutex类。某个概念的一个精炼就是这个概念的某种特殊化，如一个ReadWriteMutex就是一个带有某些附加行为的互斥体。

最后，线程只做这三件事中的某种一种：执行中、等待某个事物，或者准备执行但等待某个事物或者正在处理中。这些状态分别叫做运行*run*、等待*wait*和准备好*ready*。这些术语在以后的几节中都要用到。

12.1 创建一个线程

问题

当主线程继续它的工作时，如何创建一个新的线程来执行某项任务？

解决方案

创建类thread的一个对象，并把一个处理某项工作的函数传给它。这个thread对象的创建将实例化一个操作系统的线程，这个操作系统的线程将执行你传入的函数operator()方法（或者如果你传入的是一个函数指针的话，将从这个函数的开始处开始执行）。

示例12-1 创建一个线程

```
#include <iostream>
#include <boost/thread/thread.hpp>
#include <boost/thread/xtime.hpp>

struct MyThreadFunc {
   void operator()() {
      // Do something long-running...
   }
} threadFun;

int main() {

   boost::thread myThread(threadFun); // Create a thread that starts
                                     // running threadFun

   boost::thread::yield();           // Give up the main thread's timeslice
                                     // so the child thread can get some work
                                     // done.

   // Go do some other work...

   myThread.join(); // The current (i.e., main) thread will wait
                    // for myThread to finish before it returns
}
```

www.TopSage.com

大家网 TopSage.com

讨论

创建一个线程是比较简单的。你所需要做的就是在栈空间或堆空间上创建一个线程对象，并且把你需要做的工作的函数传给它。从讨论的角度来说，一个线程有两方面的含义。第一，它是thread类的一个对象，从传统的意义上来说它是一个C++对象。当我引用对象时，我的意思是它是一个线程对象。然后，它又是一个执行的线程，也就是说通过这个线程对象它代表了操作系统的线程。当我说线程时，我是指操作系统的线程。

让我们看一看例子中的代码。这个thread类的构造函数带一个算符(或者是函数指针)，这个算符不带参数并返回void类型。请看示例12-1中的代码：

```
boost::thread myThread(threadFun);
```

这就在栈上创建一个名为myThread的对象，它代表操作系统的一个新的线程并开始执行这个传入的函数threadFun。在这一点上，至少从理论上说，threadFun算符中的代码和main函数中的代码是并行执行的。精确地说，这两个线程并可能不是并行执行的，因为你的机器可能只有一个处理器，在这种情况下是不可能并行执行的（最近的处理器体系结构可能不是这样，但是我不考虑那些双核处理器）。如果你只有一个处理器，那么操作系统将给你创建的处在运行run状态的线程在这个线程挂起前分配一个时间片。因为这些时间片可能在大小上有差别，所以你不可能保证某个线程在某个确定的点执行。这就是多线程编程中困难的地方：多线程程序的执行是不确定的。同一个多线程程序在相同的输入的情况下，多次运行的结果可能不同。多线程中资源访问的协调就是12.2节的主题。

创建完myThread后，主线程继续运行，最少一会以后，直到它碰到如下这行代码行：

```
boost::thread::yield();
```

这行代码将使得当前线程（这个例子中是主线程）进入sleep睡眠状态，也就意味着操作系统将使用特定于操作系统的策略来交换另外一个线程或另外一个进程。yield方法的功能就是告诉操作系统当前的线程需要放弃余下的时间片。同时，一个新的线程执行线程函数threadFun。但这个thread执行完时，这个子线程就会消失。注意这个线程对象不会消失，因为它仍然是一个还处在它的生存期的C++对象。这是一个重要的特性。

这个线程对象存在于堆或栈上，就像任何其他的C++对象那样。当这些调用的代码退出了它的范围时，任何栈上的线程对象就会被销毁掉，并且当调用者使用一个指向线程的指针作为参数调用delete方法时，相应的堆上的线程对象也会被销毁掉。但是线程对象事实上仅仅是操作系统线程的代理，并且当这些线程对象被销毁掉时，操作系统线程并不能保证就消失。它们仅仅是从这个对象中分离开了，也就意味着这些线程以后是不能再join的。这点并不坏。

线程使用资源，并且在任何一个设计良好的多线程应用程序中，访问这些资源（对象、网

www.TopSage.com

络连接、文件、内存等）都需要使用互斥体 mutex 来控制，这些互斥体是一些使用来让这些线程串行化访问资源的对象（请看12.2节）。如果一个操作系统上的线程被清除掉的话，它将不释放它的锁或释放它的资源，相同地杀死一个进程也不会有机会来擦除新缓存或者正确地释放操作系统的资源。当你设想一个线程应该结束时就简单地结束一个线程就如同当一个油漆匠的时间用完时你就把它用的梯子从他下面移走一样。

因此，我们就有了join成员函数。就像在示例12-1中那样，你可以调用join方法来等着它的子线程结束。join是一个礼貌的方法，它告诉线程你正在等待直到它做完它的工作：

```
myThread.join();
```

这个调用join方法的线程进入wait等待状态直到myThread代表的线程完成为止。如果它不结束的话，join方法就不会返回。join是一个等待子线程结束的最好的方法。

你可能注意到如果你在threadFun中实现一些有意义的事情，但把join方法的使用代码注释起来的话，线程将不能完成它的工作。在这个threadFun中放入一个循环或某些比较费时的工作试试看。这是因为当操作系统销毁掉一个进程时，它的所有子线程也将随它而去，而不会管它们是否已经完成了。不调用join方法，主线程就不会等待它的子线程：主线程退出，并且操作系统意义上的线程就被销毁了。

如果你需要创建几个线程，考虑使用一个线程组对象thread_group来组织它们。一个thread_group对象可以使用多种办法来管理线程。首先，你可以使用一个指向一个线程对象的指针作为参数来调用add_thread方法，然后这个线程对象就被加入到这个线程组中。如下一个例子所示：

```
boost::thread_group grp;
boost::thread* p = new boost::thread(threadFun);
grp.add_thread(p);
// do something...
grp.remove_thread(p);
```

当这个grp的析构函数被调用时，它将删除所有这些通过add_thread方法加入的线程指针。由于这个原因，你仅能把一个堆上的线程对象的指针加入到一个线程组thread_group中。调用remove_thread方法并给这个方法传递一个线程对象的地址则从这个线程组中删除一个线程（remove_thread通过比较指针的值而不是这些指针所指向对象来在这个线程组中找到相应的指针对象）。remove_thread方法从线程组中删除某个线程的指针，但是你仍然负责把线程本身删除。

你可以调用create_thread方法，先不创建某个线程而直接把线程加入到线程组中，也就意味着这个方法（同一个线程对象一样）带一个函数算符作为参数并且创建一个新的操作系统上的线程。例如，在一个线程组中创建两个线程，如下所示：

```
boost::thread_group grp;
```

www.TopSage.com

```
grp.create_thread(threadFun);
grp.create_thread(threadFun); // Now there are two threads in grp

grp.join_all(); // Wait for all threads to finish
```

无论是你通过create_thread还是add_thread方法把线程加到线程组，你都可以调用join_all来等待这个线程组中的所有线程执行完。调用join_all方法在功能上与你在这个线程组中的每一个线程上调用join方法完成的功能一样：当这个线程组中所有线程的工作都完成时，join_all方法返回。

创建一个线程对象让一个单独的线程开始执行。使用Boost线程库这样做是非常容易的，尽管设计时还是要非常仔细。更多的有关线程的警戒信息请参考这一章的其他几节。

请参考12.2节。

12.2 使资源是线程安全的

问题

当一个程序使用了多个线程时，如何保证在同一时刻多个线程不会修改同一个资源？总的来说，这个过程被称为资源线程安全，或者是串行化访问资源。

解决方案

使用定义在*boost/thread/mutex.hpp*中的mutex类，从而使得线程对资源的访问串行化。示例12-2给出了一个简单地使用mutex对象来控制对一个队列queue的并发访问。

示例12-2 使一个类是线程安全的

```
#include <iostream>
#include <boost/thread/thread.hpp>
#include <string>

// A simple queue class; don't do this, use std::queue
template<typename T>
class Queue {
public:
   Queue() {}
  ~Queue() {}

   void enqueue(const T& x) {
      // Lock the mutex for this queue
      boost::mutex::scoped_lock lock(mutex_);
      list_.push_back(x);
      // A scoped_lock is automatically destroyed (and thus unlocked)
      // when it goes out of scope
   }

   T dequeue() {
     boost::mutex::scoped_lock lock(mutex_);

     if (list_.empty())
        throw "empty!";      // This leaves the current scope, so the
     T tmp = list_.front(); // lock is released
```

www.TopSage.com

```
      list_.pop_front();
      return(tmp);
   } // Again: when scope ends, mutex_ is unlocked

private:
   std::list<T> list_;
   boost::mutex mutex_;
};

Queue<std::string> queueOfStrings;

void sendSomething() {
   std::string s;
   for (int i = 0; i < 10; ++i) {
       queueOfStrings.enqueue("Cyrus");
   }
}

void recvSomething() {
   std::string s;

   for (int i = 0; i < 10; ++i) {
      try {s = queueOfStrings.dequeue();}
      catch(...) {}
   }
}

int main() {
   boost::thread thr1(sendSomething);
   boost::thread thr2(recvSomething);

   thr1.join();
   thr2.join();
}
```

讨论

使类、函数、代码块或者其他的对象线程安全是多线程程序设计的关键。如果你正在设计的软件是多线程的，那么每一个线程就都可能拥有它自己的一序列的资源，如栈和堆对象、操作系统资源等。但是不久你就会发现需要在线程之间共享某些资源。它可能是那些进来的工作需求（如在一个多线程web服务器上）的一个共享的队列或者是简单的一个输出流（如一个日志文件或者一个事件输出流中）。协调这个资源的安全访问的标准方法就是使用一个互斥体 *mutex*，它提供了对某个事物的排他性访问。

这个讨论的其他部分将描述总体上一个mutex是什么，并且为了串行化对一个资源的访问该如何使用`boost::mutex`。我使用前面讨论的概念/模型方法来介绍它。一个概念就是某个事物描述的抽象（不依赖于语言的），而一个概念的模型就是C++中类形式的一个具体代表。一个概念的精炼就是一个带有某些附加或扩张行为的概念。

尽管并发编程是一个复杂的主题，但是还是有很多的技术能适用于某个简单的应用中。同时又有很多不同的设计模式可以使用，并且在不同的应用中又有不同的策略可以使用。如果你需要设计一个较大的多线程程序，或者你的应用程序对性能的要求很高，你就应该多看看有关多线程设计模式的一些好的书籍。那些使得多线程的调试变得很困难的大部分问题都能通过仔细的单调一点的设计避免掉。

www.TopSage.com

使用 mutex

mutex的概念是简单的：一个mutex是一个代表一个资源的事物并且能够一次被一个线程锁定和解锁。它是一个用来协调由多个消费者使用的资源访问的标志。在Boost线程库中，这个mutex概念由boost::mutex类来实现。在示例12-2中，通过使用一个mutex的成员变量来保证对Queue类的写访问：

```
boost::mutex mutex_;
```

mutex_必须被任意的成员函数锁定，这些任意的成员函数必须改变这些被维护元素的队列的状态。这个mutex对象本身并不知道它代表的是什么。它仅仅是那些被多个消费者使用的资源的锁定/解锁的标志。

在示例12-2中，当一个Queue的成员函数需要改变这个对象的状态时，它必须首先锁定这个mutex_。在某个时刻，只有一个线程可以锁定它，这就阻止了同一时刻有多个对象可以改变这个Queue对象的状态。因此，一个mutex就是一个简单的信号机制，但它和一个简单的布尔型或整型数不同，因为一个mutex需要串行化访问，这只能由操作系统的内核来保证。如果你使用一个布尔型变量来这样做，它将达不到目的，因为没有什么东西可以阻止多个线程在同一时刻改变这个布尔型变量的状态（不同的操作系统有不同的实现，这也就是不容易实现一个可移植的线程库的原因）。

可以使用不同的锁策略来给mutex对象加锁和解锁，最简单的方法就是使用scoped_lock。一个scoped_lock是一个类，使用一个mutex参数来构造它，并且一直锁定它直到这个锁被销毁。请看示例12-2中的enqueue成员函数中scoped_lock是如何工作的：

```
void enqueue(const T& x) {
   boost::mutex::scoped_lock lock(mutex_);
   list_.push_back(x);
} // unlocked!
```

当这个锁被销毁时，mutex_就解锁。如果这个正在被构造锁的mutex已经被别的线程锁定的话，当前线程就会进入wait等待状态直到这个锁被解开。

这个设计第一眼看上去你会觉得奇怪，为什么不在mutex上设置一个lock和unlock方法呢？事实上使用scoped_lock类的方法在构造时就加锁而在析构时就解锁更为方便并且更不容易出错。当你使用scoped_lock方法创建一个锁时，它在它的整个生命周期都锁定这个对象，这就意味着你没必要在每一个控制路径上显式地解锁。另一方面，如果你需要不得不显式地锁定mutex，你就不得不捕获你的函数抛出的任意异常（或者在调用栈的函数中的任意地方）并且解锁这个mutex。而使用scoped_lock，如果有异常抛出或函数返回的话，这个scoped_lock对象都会被销毁并且这个mutex会被解锁。

使用一个mutex能使工作完成，但是还是有点美中不足。它没有区分读和写，而这是很重要的，因为线程如果只是做只读操作的话，将会强制线程串行化访问，这样效率是很

www.TopSage.com

大学网 TopSage.com

低的，而且这种读操作不需要排他性访问。基于这个原因，Boost线程库提供了read_write_mutex。示例12-3说明如何用一个让调用者在不弹出这个queue中的第一个元素时就获取这个元素的副本的成员函数front，并让这个函数与read_write_mutex一起使用实现示例12-2中的功能：

示例12-3 使用读/写mutex

```
#include <iostream>
#include <boost/thread/thread.hpp>
#include <boost/thread/read_write_mutex.hpp>
#include <string>

template<typename T>
class Queue {
public:
   Queue() :  // Use a read/write mutex and give writers priority
      rwMutex_(boost::read_write_scheduling_policy::writer_priority) {}
  ~Queue() {}

   void enqueue(const T& x) {
      // Use a r/w lock since enqueue updates the state
      boost::read_write_mutex::scoped_write_lock writeLock(rwMutex_);
      list_.push_back(x);
   }

   T dequeue() {
      // Again, use a write lock
      boost::read_write_mutex::scoped_write_lock writeLock(rwMutex_);

      if (list_.empty())
         throw "empty!";
      T tmp = list_.front();
      list_.pop_front();
      return(tmp);
   }

   T getFront() {
      // This is a read-only operation, so you only need a read lock
      boost::read_write_mutex::scoped_read_lock readLock(rwMutex_);
      if (list_.empty())
         throw "empty!";
      return(list_.front());
   }

private:
   std::list<T> list_;
   boost::read_write_mutex rwMutex_;
};

Queue<std::string> queueOfStrings;

void sendSomething() {
   std::string s;

   for (int i = 0; i < 10; ++i) {
      queueOfStrings.enqueue("Cyrus");
   }
}

void checkTheFront() {
   std::string s;

   for (int i = 0; i < 10; ++i) {
      try {s = queueOfStrings.getFront();}
```

www.TopSage.com

```
        catch(...) {}
    }
}

int main() {

    boost::thread thr1(sendSomething);
    boost::thread_group grp;

    grp.create_thread(checkTheFront);
    grp.create_thread(checkTheFront);
    grp.create_thread(checkTheFront);
    grp.create_thread(checkTheFront);

    thr1.join();
    grp.join_all();
}
```

这里有很多我应该指出的。请注意我这里使用了 read_write_mutex，如下所示：

```
boost::read_write_mutex rwMutex_;
```

当你使用读/写 mutex 时这个锁是不同的。在示例 12-3 中，当我为了写而把这个 Queue 锁定时，我创建了一个 scoped_write_lock：

```
boost::read_write_mutex::scoped_write_lock writeLock(rwMutex_);
```

并且当我仅需要读这个 Queue 时，我使用了一个 scoped_read_lock：

```
boost::read_write_mutex::scoped_read_lock readLock(rwMutex_);
```

读/写锁都是便利的，但是它们不能阻止你自己绊着自己。当你仅使用一个读锁时，为了保证你没有改变资源，它们没有对 rwMutex_ 代表的资源在编译时进行检查。由于编译器不会检查，所以当你有一个写锁时你需要采取额外的工作来保证一个线程仅修改了这个对象的状态。

精确地说这些锁的调度依赖于当你构造这个 mutex 时选定的调度策略。Boost 线程库中提供了四种策略：

reader_priority
: 等待读锁的线程将优先于那些等待写锁的线程得到锁。

writer_priority
: 等待写锁的线程将优先于那些等待读锁的线程得到锁。

alternating_single_read
: 在读锁和写锁之间交替。当某个时刻轮到了某个读操作的顺序时就允许这个简单读操作一个读锁。这个策略总的来说还是写优先的。例如，如果一个 mutex 是写锁定，并且同时有好几个挂起来的读锁和一个挂起来的写锁，那么一个读锁将会被允许，然后是一个写锁被允许，然后是所有的读锁被允许。在这段时间内假定没有新的锁请求。

www.TopSage.com

alternating_many_reads

在读锁和写锁之间交替。当某个读锁轮到了它的时刻时，允许所有的读锁。换句话说，这个策略将在两个写锁之间使得所有的在这个queue上挂起来的读锁都被允许。

这些策略都有各自的优点和缺点，并且它们的执行都依赖于你的应用程序。决定使用哪种策略需要仔细考虑，因为简单地使用读锁或写锁都可能会导致某些锁不能成功也即是饿死的出现，下面我将要详细地描述。

危险

当你设计一个多线程程序时，通常有三个基本问题：死锁、饿死和竞态条件。有很多技术可以用来避免它们的出现，这些技术的复杂性是不同的，在本节中就不准备讨论它们的复杂性。我将详细地描述每一个这种问题，这样你就能知道该如何提防它们，但是如果你打算开发多线程程序的话，你还是应首先看看多线程设计模式。

死锁是涉及到最少两个线程和两个资源的某种情况。假设两个线程A和B，两个资源X和Y，线程A锁定了资源X，而线程B锁定资源Y。当A又想锁定资源Y而线程B又想锁定资源X时，死锁就出现了。如果线程没有设计打破一定的死锁的话，那么它们将永远等待下去。

Boost线程库通过对mutex和锁概念的精炼让你能避免死锁。一个试探性的mutex支持对锁的试探性操作，使用一个试探性的锁可能成功也可能失败，但不会被阻塞住并一直等到资源被锁住。try_mutex和scoped_try_lock就是这个概念的模型形式，你的代码在你需要的资源被锁住时不会阻塞住并且可以从这个锁出来去做一些别的操作。还有另一种关于这个试探性锁的概念的精炼就是时间锁。使用一个时间锁，当阻塞的时间到达某个特定的数时线程就会放弃。这里关于时间锁的详细技术就不讨论了，可以参考Boost线程库文档。

例如，示例12-2中的Queue类，你想使用一个试探性的mutex，因此dequeue返回一个布尔型值来表明是否成功地获取了第一个元素。这样dequeue方法的使用就不必非得一直等待直到这个queue被锁定。下面就是实现这个dequeue的例子：

```
bool dequeue(T& x) {
   boost::try_mutex::scoped_try_lock lock(tryMutex_);

   if (!lock.locked())
      return(false);
   else {
      if (list_.empty())
         throw "empty!";
      x = list_.front();
      list_.pop_front();
      return(true);
   }
}
private:
```

www.TopSage.com

```
boost::try_mutex tryMutex_;
// ...
```

使用的mutex和锁与示例12-2中的都不同。请正确地保证你正在使用的mutex和锁的类的名字，否则，你将不能得到你期望的行为。

当串行化访问某个事物时，你就告诉用户排队并且等待他自己的顺序。如果每个人都站在队中相同的位置，那么每个人都会得到机会使用这个资源。但如果你让别人插队的话，就有可能使得队列后面的人永远也得不到机会使用资源。这就是饿死。

当使用一个mutex时，用户是在一个组而不是在一个队列中等待。这些等待锁的线程先后顺序是没有保证的。对读/写mutex来说，Boost线程库使用了前面介绍的四种调度策略。因此，当使用读/写mutex时，必须意识到这些不同的调度策略是什么意思，并且需要明白线程正在做什么。如果你正在使用writer_priority策略，并且你有很多创建写锁的线程，那么你的读锁的线程就可能饿死；相同的情况reader_priority也可能发生，因为这些调度策略总是优先于某种类型的锁的。通过测试，如果你意识到了某种类型的线程不能达到它应该有的程度的话，你就应该考虑使用alternating_many_reads或alternating_single_read策略。当你构造一个读/写的mutex时你就指定了策略。

最后，一个竞态条件就是这种情况：你的代码对锁的顺序或锁的原子性做出的某种假设是错误的。例如，考虑Queue类的一个消费者在这个queue的头部查询并有条件地调用dequeue：

```
if (q.getFront() == "Cyrus") {
   str = q.dequeue();
   // ...
```

这些代码在单线程的环境中工作得很好，因为q在第一行代码和第二行代码之间不会被修改。然而，当使用多线程时，你就不得不考虑这种情况：别的线程可能在任意时刻修改q，事实上当一个线程没有给共享对象上锁时你就应该假设这个共享对象会被修改。在第一行代码后面，另外一个线程可能进来并且从q中的下一个元素处调用dequeue，这也就意味着第二行代码可能会得到一个不可靠的结果或者什么也得不到。为了修改q，getFront和dequeue都会锁定这个简单的mutex，但在它们解锁之前，如果有另外一个线程正等待着这个锁，那么这个线程可能在第二行代码运行前就把锁获取过去了。

对这种竞态条件的一个解决方案就是保证在某个操作的全过程中都持有锁。创建一个名为dequeueIfEquals的成员函数，它仅仅在等于它的输入参数的那个元素上调用dequeues。dequeueIfEquals可以使用锁：

```
T dequeueIfEquals(const T& t) {
   boost::mutex::scoped_lock lock(mutex_);
   if (list_.front() == t)
      // ...
```

www.TopSage.com

还有其他类型的竞态条件，但这个例子给你指出了一个该如何提防它的总体意见。随着你使用的线程和共享资源的增加，这些竞态条件将变得更为敏感和难以捕获。因此，你应该采取一些特殊考虑的设计来阻止它的发生。

多线程开发中保证资源的串行化访问是一个最难的课题，因为当你没有正确的处理时，调试是非常困难的。由于多线程程序天生就具有不确定性（因为线程的执行具有不同的顺序并且每次程序的执行都具有不同的时间），因此要精确地查明一个线程在哪里或者是如何修改了它不应该修改的事物是非常痛苦的。将会比单线程编程花费更多的工作，一个好的设计就是使得调试和重复工作最少。

12.3 从一个线程中给另一个线程发送通知

问题

你正在使用一个模式，在这个模式中一个线程（或者一组线程）处理某些事情并且这个线程需要让别的线程（或别的线程组）知道它。你可以有一个负责分发工作给别的辅线程的主线程，或者你可能使用一组线程来向一个队列中追加数据，并且另一个线程从这个队列中取数据然后做一些有用的事情。

解决方案

使用定义在 *boost/thread/mutex.hpp* 和 *boost/thread/condition.hpp* 文件中的 mutex 和 condition 对象。当你需要线程等待某个事物时你就可以创建一个 condition 对象，然后通过这个 condition 对象来通知那些等待的线程。示例 12-4 说明了如何在一个主/辅线程模型中发送信号。

示例 12-4 在线程之间发送信号

```
#include <iostream>
#include <boost/thread/thread.hpp>
#include <boost/thread/condition.hpp>
#include <boost/thread/mutex.hpp>
#include <list>
#include <string>

class Request { /*...*/ };

// A simple job queue class; don't do this, use std::queue
template<typename T>
class JobQueue {
public:
   JobQueue() {}
  ~JobQueue() {}

   void submitJob(const T& x) {
      boost::mutex::scoped_lock lock(mutex_);
      list_.push_back(x);
      workToBeDone_.notify_one();
   }
```

www.TopSage.com

大家网 TopSage.com

```
    T getJob() {
      boost::mutex::scoped_lock lock(mutex_);

      workToBeDone_.wait(lock); // Wait until this condition is
                                // satisfied, then lock the mutex
      T tmp = list_.front();
      list_.pop_front();
      return(tmp);
    }

private:
    std::list<T> list_;
    boost::mutex mutex_;
    boost::condition workToBeDone_;
};

JobQueue<Request> myJobQueue;

void boss ) {
    for (;;) {
      // Get the request from somewhere
      Request req;
      myJobQueue.submitJob(req);
    }
}

void worker() {
    for (;;) {
      Request r(myJobQueue.getJob());
      // Do something with the job...
    }
}

int main() {
    boost::thread thr1(boss);
    boost::thread thr2(worker);
    boost::thread thr3(worker);

    thr1.join();
    thr2.join();
    thr3.join();
}
```

讨论

一个condition对象使用mutex对象，并且让你等待这个mutex对象不被锁定这种情况。考虑示例12-4，它是示例12-2中说明的Queue类的一个改版。我把这个Queue类修改成某个特殊形式的队列类，叫做JobQueue类，在这个类中，对象代表由主线程提交的工作并且这个工作由辅线程获取。

这个JobQueue类的最重要的变化就是这个condition类型的成员变量workToBeDone_。这个condition对象表明在这个队列中是否有工作存在。当一个线程想从这个工作队列中获取一个元素时，它就需要调用getJob方法，这个方法需要获取这个mutex互斥对象上的一个锁，然后等待取到这个锁，如下所示：

```
    boost::mutex::scoped_lock lock(mutex_);
    workToBeDone_.wait(lock);
```

第一行通过一种平常的方法来锁定这个mutex互斥对象。第二行代码解开这个mutex上

www.TopSage.com

的锁然后进行等待或者休眠，直到它的条件得到了满足。这个mutex互斥对象的解锁让其他的线程能够使用这个mutex对象，它们中的某个需要设置这个等待的条件，否则，其他的线程将不能锁定这个正在被某个线程等待的mutex互斥对象。

在这个submitJob方法中，当这个工作加入到内部的链表中时，我加上了如下这一行代码：

```
workToBeDone_.notify_one();
```

这就满足了getJob正在等待的条件。从技术上来说，这就意味着如果有任何在这个condition对象上调用wait的线程的话，这些线程中的某一个就会进入到运行run状态。在getJob中，也就意味着线程将继续从以下这行代码执行：

```
workToBeDone_.wait(lock);
```

但不仅仅是这样，wait方法做两件事：它一直等待直到有人在它正等待的condition上调用了notify_one或notify_all，然后它就锁定跟它相关的mutex。当submitJob方法调用notify_all时事实上发生的是正在等待的线程进入运行状态，并且接着做的就是尽量去锁定这个由submitJob方法已经锁定的mutex对象，因此它就仍进入wait等待状态直到submitJob方法完成。因此，condition::wait方法需要当mutex被调用时锁定它，在mutex对象被解锁时，如果碰到一个condition它又会被锁定。

调用notify_all时将通知那些所有正在等待某个条件变成真的线程。它和notify_one的工作方式相同，只不过那些所有正处在等待这个条件的线程都变成运行状态之外。尽管它们都尽量去获取下一个锁，接下来会发生什么依赖于这个mutex的类型和使用的锁的类型。

一个condition能使你达到一个微妙的效果，也就是当你使用mutex并单独锁定它们时，你并不能取得它。考虑一下前面的那个Queue类。正在等待双端队列的线程一直等待直到它获得一个写锁，然后从这个队列中弹出下一个元素。不使用任何类型的信号机制也许这个类能工作得很好，但真的是这样吗？当这个队列是空的时候呢？如果你正在等待某个条件变成真的话也许你有很多种选择来实现这个双端队列：获取一个锁；查看这个队列中是否有元素，如果没有，就返回；使用另一个mutex，当这个队列为空时就锁定这个mutex，而当这个队列有元素时就给这个mutex解锁（这不是一个好的想法）；或者当这个队列为空时就返回一个特殊的值。这几种方法要么有问题要么就效率不高。如果这个队列为空时通过抛出一个异常或返回一个特殊的值而简单地返回，那么你的客户将不得不不断地检查是否有东西到达。这在资源上是一个不必要的消耗。

一个condition对象能让消费者线程休眠，因此在还没有碰到一个condition时处理器就可以去处理别的事情。例如一个web服务器使用一个工作线程池来处理进来的请求。当没有需求进来时让这些子线程处于等待状态比让它们循环地查询或睡眠然后偶尔唤醒来检查这个队列要好得多。

www.TopSage.com

12.4 只初始化一次共享资源

问题

有一序列的线程，如何使得它们能使用那些只能初始化一次的资源？

解决方案

在线程使用这个资源前就初始化这个资源，或者如果你不能这样做的话，就调用定义在<boost/thread/once.hpp>中的call_once函数。示例12-5说明该如何使用这个call_once函数。

示例12-5 只初始化一次某个资源

```
#include <iostream>
#include <boost/thread/thread.hpp>
#include <boost/thread/once.hpp>

// Some sort of connection class that should only be initialized once
struct Conn {
   static void init() {++i_;}
   static boost::once_flag init_;
   static int i_;
   // ...
};

int Conn::i_ = 0;
boost::once_flag Conn::init_ = BOOST_ONCE_INIT;

void worker() {
   boost::call_once(Conn::init, Conn::init_);
   // Do the real work...
}

Conn c;  // You probably don't want to use a global, so see the
         // next Recipe

int main() {

   boost::thread_group grp;

   for (int i = 0; i < 100; ++i)
 grp.create_thread(worker);

   grp.join_all();

   std::cout << c.i_ << '\n'; // c.i_ = 1
}
```

讨论

一个共享资源不得不再在某个地方被初始化，并且你可能期望第一次使用这个资源的线程来完成它的初始化工作。一个once_flag类型（它的准确类型由平台本身决定）和这个call_once函数能够保证多个线程不会重复地初始化同一个对象。但是你不得不做两件事。

www.TopSage.com

首先，使用BOOST_ONCE_INIT宏来初始化这个once_flag类型的对象。这个宏的具体值由平台本身决定。示例12-5中，类Conn代表某种类型的连接（数据库、网络套接字、硬件等），我期望这个连接只初始化一次，尽管多个线程可能都会初始化它。常常当你想动态地装载一个库时这种情况就会出现，这个库可能是在一个应用程序的配置文件中指明的。这个once_flag是一个静态类变量，因为我仅期望初始化一次，而不管这个类可能有多个实例。因此，我给这个变量赋一个BOOST_ONCE_INIT值作为起始值，如下所示：

```
boost::once_flag Conn::initFlag_ = BOOST_ONCE_INIT;
```

然后，在我的工作函数中，我调用call_once函数，它同步对这个initFlag_变量的访问，这样就禁止了对这个变量的并发访问。我给这个函数传递了两个参数：

```
boost::call_once(Conn::init, Conn::initFlag_);
```

第一个参数用来做初始化函数的地址。第二个参数就是这个标志。这样的话，多个线程都可以初始化这个Conn类，但是只有一个会成功。

12.5 给线程函数传递一个参数

问题

在Boost线程库中的创建函数只接受不带参数的算符，如何实现给你的线程函数传递一个参数？

解决方案

创建一个带着你的参数的算符适配器并返回一个不带参数的算符。当你需要给线程函数传递一个不带参数的算符时就可以使用这个算符适配器。请看一下示例12-6是如何这样做的：

示例12-6 给一个线程函数传递一个参数

```
#include <iostream>
#include <string>
#include <functional>
#include <boost/thread/thread.hpp>

// A typedef to make the declarations below easier to read
typedef void (*WorkerFunPtr)(const std::string&);

template<typename FunT,    // The type of the function being called
         typename ParamT> // The type of its parameter
struct Adapter {
   Adapter(FunT f, ParamT& p) : // Construct this adapter and set the
      f_(f), p_(&p) {}          // members to the function and its arg

   void operator()() { // This just calls the function with its arg
      f_(*p_);
   }
```

www.TopSage.com

```
private:
   FunT    f_;
   ParamT* p_;  // Use the parameter's address to avoid extra copying
};

void worker(const std::string& s) {
   std::cout << s << '\n';
}

int main() {

   std::string s1 = "This is the first thread!";
   std::string s2 = "This is the second thread!";

   boost::thread thr1(Adapter<WorkerFunPtr, std::string>(worker, s1));
   boost::thread thr2(Adapter<WorkerFunPtr, std::string>(worker, s2));

   thr1.join();
   thr2.join();
}
```

讨论

你需要解决的基本问题并不特定于线程或者Boost线程库，当你不得不传递一个某种签名的算符给某个需要一个与它不同签名的算符作为参数的算法时，这是一个通常的问题。这个问题的解决方案就是创建一个适配器。

这个语法可能有点乱，但示例12-6中必须做的就是创建一个临时的算符，这个临时的算符就像人们期望的那样可被一个线程构造函数作为一个不带参数的函数而调用。第一件事情就是：使用一个typedef来重新定义一下这个函数指针使得它更容易阅读：

```
typedef void (*WorkerFunPtr)(const std::string&);
```

这就创建了一个WorkerFunPtr类型，它是一个函数指针，它带一个字符串引用作为参数并且返回一个void类型。然后，我创建这个适配器类模板。它提供了实例化一个动态算符的方法。请看下面这个构造函数：

```
template<typename FunT,
         typename ParamT>
struct Adapter {
   Adapter(FunT f, ParamT& p) :
      f_(f), p_(&p) {}
// ...
```

这个构造函数所做的就是实例化两个成员，它们可以是任何类型，但我们期望它们是一个函数指针和任何类型的参数p。为了提高效率我把这个参数的地址存储下来，而不是采用值复制的方式来复制它。

请看来自主线程中的下面的这行代码：

```
boost::thread thr1(Adapter<WorkerFunPtr, std::string>(worker, s1));
```

这个thr1的构造函数的参数就是这个适配器类模板的一个实例，使用两个类型WorkerFunPtr和std::string作为它的参数。这个实例为适配器的f_和p_成员使

www.TopSage.com

大家网 TopSage.com

用这两个类型。最后，适配器重写了操作符()，因此它能够像函数一样被调用。当它被调用时，其形式如下所示：

```
f_(*p_);
```

使用这个适配器类模板，你就可以给线程函数传递参数，尽管需要使用一些额外的语法。如果你需要传递多个参数，仅需要在这个适配器中增加另一个类型和成员变量。这种方法的一个比较好的方面就是你可以创建一序列的泛型适配器类模板，并且你可以在各种其他的上下文中使用它们。

第13章

国际化问题

13.0 概述

本章将描述有关C++程序国际化时的几个共同问题的解决方案。使得程序能够在不同的环境中可以工作（常常称为本地化）常常需要解决两个问题：以遵循当地传统的方式格式化用户可见的字符串（如日期、时间、钱和数字），并且使得数字在不同的字符集中一致。这一章我们主要处理第一个问题，对第二个问题只做概要性的介绍，因为在不同的字符集上还很少有标准可以支持，大部分都是依赖于实现的。

大部分程序都需要在很多国家而不是只局限在实现这个程序的国家运行。为了支持这个现实，为使得实现的代码可以在不同的国家运行，C++类库提供了几个工具。然而，这几个工具的设计不同于其他的标准类库工具，如字符串、文件输入和输出、容器、算法等等。例如，用来代表一个本地环境的类是`locale`，它在头文件`<locale>`中提供。这个`locale`类为使用本地特殊的格式从流中读或向流中写提供了一些方便之处，当然也为获取一个本地环境的信息如货币符号或者日期格式提供了一些方便的方法。这个标准仅需要提供一个简单的locale，并且这个简单的locale可以是“C”或传统的locale。这个传统的locale使用ANSI C惯例：美国英语惯例和7位的ASCII字符编码。对于各种语言和地区是否需要提供`locale`实例则由实现自己来决定。

这个`<locale>`头文件有三个基本的部分。首先，这有一个`locale`类。它封装了C++支持的这个locale类行为的各个方面，并且这也是你需要做一些与某个locale相关的格式时的入口点。第二，一个locale的最小粒度部分和你与之工作的具体类是*facet*。一个facet的例子是一个类，如给一个流写入日期的`time_put`类。第三，每个facet都属于某个类型，它是用来把相关的facet组织起来的方法。这些类型的例子是数字、时间和货币（前面我提到的`time_put` facet就属于时间类型）。本章我主要讨论这些类型，但是这些类型仅当你在某个locale上作一些复杂的处理时才真的方便，因此我不会进一步讨论它们。

www.TopSage.com

每一个C++程序至少有一个locale，这个locale是当成全局locale的（它经常是作为一个全局的静态对象实现的）。默认的情况下，它是传统的“C” locale，除非你把它改变成别的locale了。locale的其中一个构造函数可以让你初始化用户需要的locale，尽管你可以很自由地精确定义什么是用户需要的locale。

在大部分情况下，当从流中读或写时，你仅需要使用locale类。这是本章的焦点。

13.1 硬编码Unicode字符串

问题

如何硬编码一个Unicode，如宽字符、源文件中的字符串？

解决方案

通过使用一个前置L来硬编码这个字符串并且把这个字符像任何别的字符串一样输入到你的源文件编辑器中，或者使用十六进制数字来代替这个unicode字符。示例13-1说明这两种办法。

示例13-1 硬编码一个Unicode字符串

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main() {

   // Create some strings with Unicode characters
   wstring ws1 = L"Infinity: \u221E";
   wstring ws2 = L"Euro: _";

   wchar_t w[] = L"Infinity: \u221E";

   wofstream out("tmp\\unicode.txt");
   out << ws2 << endl;
   wcout << ws2 << endl;
}
```

讨论

硬编码一个Unicode字符串的大部分事情是如何在你的源文件编辑器中输入这个字符串。C++提供了一个宽字符类型wchar_t，它可以用来存储这个Unicode字符串。这个类型的实现是预先定义好的，常常是使用UTF-32编码。wstring类（定义在<string>中）是一个wchar_t字符的序列，就像string类是char字符的序列一样（当然，严格来说，wstring是basic_string<wchar_t>的typedef）。

输入一个Unicode字符的最容易的方法是在一个字符串的前面前置L，如示例13-1所示：

www.TopSage.com

```
wstring ws1 = L"Infinity: \u2210";  // Use the code itself
wstring ws2 = L"Euro: _";           // Or just type it in
```

现在，你可以把这些宽字符串写到一个宽字符流中去，如下所示：

```
wcout << ws1 << endl; // wcout is the wide char version of cout
```

当然，这也适用于文件：

```
wofstream out("tmp\\unicode.txt");
out << ws2 << endl;
```

处理不同字符编码最棘手的部分不是把你正确的字符加到你的源文件中，而是知道从一个数据库、HTTP需求、用户输入等得到数据如何知道它是什么类型的字符集，但是这超过了C++标准的讨论范围。C++标准并不需要特定的编码，你的操作系统用来存储源文件的字符编码可以是任意类型，只要它支持最少的由C++支持的96个字符就可以。由于这些字符不是字符集的一部分，因此它们叫做基本源字符集，标准标明它们通过使用 *\uXXXX* 或 *\UXXXXXXXX* 转义序列是必须可以得到的，这里每一个 X 都是十六进制数字。

13.2 读取数字

问题

如何以一种遵循本地传统的格式化方式把一个数字写到一个流中去？这种方式可能相互不同，并且依赖于你所在的位置。

解决方案

就像在示例13-2中那样，使用当前的locale来贯通当前你正在写入的流，或者使用你可以设置这个全局的locale然后再创建一个流。讨论部分将讨论后者。

示例 13-2 使用本地格式写入数字

```
#include <iostream>
#include <locale>
#include <string>

using namespace std;

// There is a global locale in the background that is set up by the
// runtime environment. It is the "C" locale by default. You can
// replace it with locale::global(const locale&).
int main() {
   locale loc(""); // Create a copy of the user's locale
   cout << "Locale name = " << loc.name() << endl;

   cout.imbue(loc); // Tell cout to use the formatting of
                    // the user's locale

   cout << "pi in locale " << cout.getloc().name() << " is "
        << 3.14 << endl;
}
```

www.TopSage.com

讨论

示例13-2说明了该如何使用用户的locale来格式化一个浮点型数字。这样做需要两步，创建一个本地locale类的实例，然后再把它与流相关联。

作为开始，示例13-2创建loc，它是这个用户的locale的副本。你不得不以一个空字符串来使用locale的构造函数来这样做（并不是这个默认的构造函数），如下所示：

```
locale loc("");
```

这个区别是微妙的但是很重要，后面将会来讨论。按照这种方式来创建一个locale对象创建这个用户的locale的副本，这个用户的locale是实现时定义的。这就意味着如果一台机器配置的是使用美国英语，locale::name()将返回一个locale字符串，如“en_US”、“English_United States.1252”、“english-american”等。这些事实上的字符串是实现时定义的，并且C++标准的唯一要求就是“C”。

为了比较，locale的默认构造函数返回当前全局locale的副本。对一个运行的C++程序，只有单个全局的locale对象（在一些运行库中，可能是作为一个静态变量实现的——精确地说，如何实现是实现定义的）。默认的情况下，它是C locale，并且你可以使用locale::global(locale& loc)来代替它。当流被创建时，它们使用这个全局的locale，这就意味着，cin、cout、cerr、wcin和wcerr都是使用C locale，因此如果你需要使用遵循某个特定的locale传统的格式时，你必须显式地改变它。

Locale的名字是非标准化的。通常，它们看上去与下面的相同：

```
<language>_<country>.<codepage>
```

这里语言或许是一个全名，如西班牙语“Spanish”，或者是两个字符的代码，如“sp”；国家或者是国家的全名，如“Colombia”；或者是两个字符的国家代码，如“CO”，并且代码页就是这个代码页，如1252。语言是唯一的必须部分。在各种系统中，使用显式的locale来做试验，在不同的编译器上搞清楚不同名字的意义。如果你使用的locale名字是不合法的，它将抛出一个运行时错误。示例13-3中给出了几个显式的locale名字的例子。

示例13-3 显式地命名locale

```
#include <iostream>
#include <fstream>
#include <locale>
#include <string>

using namespace std;

int main() {
   try {
      locale loc("");
      cout << "Locale name = " << loc.name() << endl;

      locale locFr("french");
      locale locEn("english-american");
      locale locBr("portuguese-brazilian");
```

www.TopSage.com

大家网 TopSage.com

```
        cout.imbue(locFr); // Tell cout to use French formatting

        cout << "3.14 (French) = " << 3.14 << endl;
        cout << "Name = " << locFr.name() << endl;

        cout.imbue(locEn); // Now change to English (American)

        cout << "3.14 (English) = " << 3.14 << endl;
        cout << "Name = " << locEn.name() << endl;

        cout.imbue(locFr); // Tell cout to use Brazilian formatting

        cout << "3.14 (Brazil) = " << 3.14 << endl;
        cout << "Name = " << locBr.name() << endl;
    }
    catch (runtime_error& e) {
        // If you use an invalid locale name, a runtime_error exception
        // is thrown.
        cerr << "Error: " << e.what() << endl;
    }
}
```

这个程序在 Visual C++ 7.1 编译的 Windows 平台上的输出如下所示：

```
Locale name = English_United States.1252
3.14 (French) = 3,14
Name = French_France.1252
3.14 (English) = 3.14
Name = English_United States.1252
3.14 (Brazil) = 3,14
Name = Portuguese_Brazil.1252
```

你可以看到我的机器的locale是使用代码页1252的美国英语。这个例子也说明了在很多其他的 locale 也使用 pi。注意，法国和巴西使用一个逗号代替小数点。千位分割符是也是不同的：法语和葡萄牙语使用空格符来代替逗号分割符，因此，在美国英语中的1 000 000.25 在法语和葡萄牙语中写法为 1 000 000,25。

在大部分情况下，你都应该避免显式地使用名字来创建locale。对于使用locale来打印数字、日期、货币或者其他别的东西，你应该简单地使用一个空字符串来实例化一个locale，然后用它来调用 imbue 成员函数填充你的流。

locale 的行为可能还是有点难以理解，因此我总结了几个重要的关键点：

- 默认的全局 locale 是“C”locale，因为它是在每一个实现中由标准得以保证的。
- 在程序开始时，标准流都是使用全局 locale 即“C”locale 来创建的。
- 通过传递一个空字符串给这个locale构造函数，如`locale("")`，你就可以创建一个用户运行时的 `locale` 的副本。
- 通过传递一个标识 locale 的字符串，你就可以创建一个命名的 `locale` 对象，如 `locale("portuguese-brazilian")`。尽管这个字符串是没有标准化的。
- 一旦你有一个代表用户的默认的 `locale` 对象或者是一个命名的locale对象，你就可以使用 `locale::global` 来设置这个全局的locale。随后创建的所有流都使用这个全局的 locale。

www.TopSage.com

- 你可以使用 imbue 成员函数来显式地给一个流设置一个 locale。

当你写的软件使用 locale 时，为用户可见的数据使用本地化的格式。也就是说，你需要使用用户熟悉的格式显示数字，实例化一个 locale，然后用这个实例来调用 imbue 成员函数填充这个流从而给用户正确地显示数字。但是如果你正在把数据写到一个文件中或者其他中间顺序化存储介质中，为了可移植性，请使用“C” locale。如果你的代码显式地改变了全局的 locale，然后你需要显式地使用“C”来显式地调用 imbue 成员函数来填充你的文件流。你可以有两种方法来这样做，通过使用名字“C”来创建一个 locale，或者调用 locale::classic()函数。如下所示：

```
ofstream out("data.dat");
out.imbue(locale::classic());
out << pi << endl; // Write using C locale
```

读数字是相同的。例如，在法语中读数字并且使用“C” locale 写数字，如下所示：

```
double d;
cin.imbue(locale("french"));
cin >> d;
cout << "In English: " << d;
```

如果你运行这个程序并且输入 **300,00**，它将打印出 300。

为了使得一个流遵循一个 locale 的数字传统，使用这个目标的 locale 对象来显式地调用 imbue 成员函数来填充这个流。或者，如果你希望所有创建的流使用某个特定的 locale，可以把它作为全局 locale 安装它。货币处理起来稍有不同；关于如何读写货币，请参考 13.4 节中的例子。

请参考 13.4 节。

13.3 读取日期和时间

问题

如何用本地的格式习惯来显示或读取时间和日期？

解决方案

为了读取日期和时间，可以使用来自 <ctime> 的 time_t 类型和 tm 结构以及来自 <locale> 中的时间和日期 facet（一会之后就会讨论 facet）。请看示例 13-4 中的例子。

示例 13-4 读取日期

```
#include <iostream>
#include <ctime>
#include <locale>
#include <sstream>
#include <iterator>
```

www.TopSage.com

```
using namespace std;

void translateDate(istream& in, ostream& out) {

   // Create a date reader
   const time_get<char>& dateReader =
     use_facet<time_get<char> >(in.getloc());

   // Create a state object, which the facets will use to tell
   // us if there was a problem.
   ios_base::iostate state = 0;

   // End marker
   istreambuf_iterator<char> end;

   tm t; // Time struct (from <ctime>)

   // Now that all that's out of the way, read in the date from
   // the input stream and put it in a time struct.
   dateReader.get_date(in, end, in, state, &t);

   // Now the date is in a tm struct. Print it to the out stream
   // using its locale. Make sure you only print out what you
   // know is valid in t.
   if (state == 0 || state == ios_base::eofbit) {
     // The read succeeded.
     const time_put<char>& dateWriter =
       use_facet<time_put<char> >(out.getloc());

     char fmt[] = "%x";

     if (dateWriter.put(out, out, out.fill(),
                        &t, &fmt[0], &fmt[2]).failed())
         cerr << "Unable to write to output stream.\n";
   } else {
     cerr << "Unable to read cin!\n";
   }
}

int main() {

   cin.imbue(locale("english"));
   cout.imbue(locale("german"));
   translateDate(cin, cout);
}
```

这个程序的输出如下：

```
3/28/2005
28.03.2005
```

讨论

读取日期和时间数据需要一些关于locale设计细节的知识。如果你还不熟悉locale和facet概念的话，请看一下13.0节。

C++并没有标准的类来代表时间和日期，与时间和日期最接近的就是来自<ctime>中的time_t类型和tm结构。如果你需要使用标准类库中的方法来读写日期，你将不得不把任何代表日期的非标准的形式转换成tm结构。这样做是值得的，因为你正在使用的实现很有可能给对于格式locale敏感的日期提供一些内在的支持。

www.TopSage.com

前面，我说明了一个facet是一个locale的一方面，它需要特定于locale的行为。更精确地说，一个facet是为了某个字符类型的模板类的一个常量型实例，它查询基于你在构造函数传入的locale类上它是如何运行的。示例13-4中，我创建了这个time_get facet的一个实例，如下所示：

```
const time_get<char>& dateReader =
  use_facet<time_get<char> >(in.getloc());
```

函数模板use_facet给一个给定的locale查找一个给定的facet。所有的这些标准的facet都是类模板，它接受一个字符类型的参数，并且由于我正读取和写入字符，我就为这些字符实例化了time_get类。这个标准要求一个实现为char和wchar_t类型提供模板的特例，因此就能保证它们能够退出来（尽管不能保证支持除了C locale之外的某个locale）。我创建的这个time_get对象是一个常量，因为由一个实现提供的locale功能是一组规则，这些规则是为了在不同的locale中格式化各类数据使用的，并且这些规则用户是不可编辑的，因此，一个给定的facet的状态是不应该被用户代码修改的。

我传给use_facet的这个locale是与我正要写入的流相关联的。getloc()声明在ios_base中并返回与一个输入或者输出流相关联的locale。使用这个已经与你要读入或者写入的流相关联的locale是最好的办法；以locale名字作为参数传递或者以其他别的办法指定都是容易出错的。

一旦我创建了马上要做事实上的读操作的对象，我就需要创建某些东西来捕获这个流的状态：

```
ios_base::iostate state = 0;
```

Facet不会自己修改流的状态，例如，set stream::failbit = 1；相反，它们将会在你的状态对象中设置状态来表明一个日期是不能读得。这是因为读取一个格式化值的失败不必一定会在流上产生问题——这个字符的输入流仍然可以是有效的——但是以你希望的格式读取它或许就不可能了。

这个事实上的日期信息存储在tm结构中，所有我不得不做的是创建一个本地的tm变量并且把它的地址传给time_get或time_put facet。

一旦我需要读取日期，我就能够查看这个我用来验证是否所有的这些都是以我希望的方式发生的状态变量。如果它的值等于0或者是ios_base::eofbit，那么这就表明这个流的状态是好的并且读出来的日期没有问题。在示例13-4中，由于我需要把日期写到另一个流中，我就不得不为这个目的创建一个这样的对象。如下所示：

```
const time_put<char>& dateWriter =
   use_facet<time_put<char> >(out.getloc());
```

它和前面的time_get类具有相同的工作方式，只是方向不同而已。之后，我创建了一个格式字符串（使用于类printf格式语法），这个格式字符串将会打印日期。“%x”打

www.TopSage.com

印日期，并且“%X”打印时间。尽管这样，你还是应该仔细：这个例子仅是读取日期，因此这个不得不处理时间的tm结构的成员在这一点上是没有定义的。

现在，我可以写入到输出流。如下所示：

```
if (dateWriter.put(out,    // Output stream iterator
    out,                   // Output stream
    out.fill(),            // Fill char to use
    &t,                    // Addr of tm struct
    &fmt[0],               // Begin and end of format string
    &fmt[2]
    ).failed())            // iter_type.failed() tells us if
                           // there was a problem writing
```

time_put::put把日期写到你传入的输出流中，它（这个time_put对象）是使用这个locale来创建的。time_put::put返回一个ostreambuf_iterator迭代器，它有一个成员函数failed，你可以调用它来查看这个迭代器是否处在一个被损坏的状态中。

get_date并不是你可以用来从一个流中得到一个日期的各个部分的唯一成员函数。还有很多别的函数：

- get_date：使用一个locale中的格式规则来从一个流中获取日期。
- get_time：使用一个locale中的格式规则来从一个流中获取时间。
- get_weekday：获取一周中的每个名字，例如，星期一等。
- get_year：使用一个locale中的格式规则来从一个流中获取年份。

其他的你觉得方便的应该就是date_order成员函数。它返回一个枚举（在<locale>中的time_base::dateorde），这个枚举值表明在一个日期中的月、天和年的顺序。如果你不得不解析time_get::put函数输出的日期，这是有用的。示例13-5说明了该如何验证这个日期的顺序。

示例13-5 查找日期顺序

```
#include <iostream>
#include <locale>
#include <string>

using namespace std;

int main() {

   cin.imbue(locale("german"));

   const time_get<char>& dateReader =
     use_facet<time_get<char> >(cin.getloc());

   time_base::dateorder d = dateReader.date_order();

   string s;

   switch (d) {
   case time_base::no_order:
     s = "No order";
```

www.TopSage.com

大家网 TopSage.com

```
      break;
    case time_base::dmy:
      s = "day/month/year";
      break;
    case time_base::mdy:
      s = "month/day/year";
      break;
    case time_base::ymd:
      s = "year/month/day";
      break;
    case time_base::ydm:
      s = "year/day/month";
      break;
    }

    cout << "Date order for locale " << cin.getloc().name()
         << " is " << s << endl;
}
```

当你初始化一个facet时，你还有一个方便的特性可以使用，那就是has_facet。它是一个函数模板，并且返回一个布尔值，这个布尔值表明这个需要的facet是否是为某个给定的locale定义的。因此，为安全起见，无论任何时候，当你需要初始化一个facet时都应利用这个has_facet。如果它返回false，你就可以一直使用默认的传统C的locale，因为它这个传统C的locale由遵循标准的实现保证了。has_facet的声明如下：

```
if (has_facet<time_put<char> >(loc)) {
   const time_put<char>& dateWriter =
     use_facet<time_put<char> >(loc);
```

一旦你熟悉了time_get和time_put类的语法，你就会发现它们使用起来是那么的直接。与以往一样，你也可以使用typedef来定义从而减少难看的尖括号的数量：

```
typedef time_put<char> TimePutNarrow;
typedef time_get<char> TimeGetNarrow;
// ...
const TimeGetNarrow& dateReader = use_facet<TimeGetNarrow>(loc);
```

以某个特定的locale格式来写或读日期是有点乏味的，但一旦你理解了你希望的locale后，它就是有效的和强大的。第5章有一个完整的关于日期和时间的主题，因此更多的关于时间和日期写入的信息，请参考5.2节。

请参考第5章和5.2节。

13.4 读取货币值

问题

如何从一个流中读或写一个已格式化的货币值？

解决方案

使用money_put和money_get facet来写和读货币值，如示例13-6所示。

示例 13-6 写和读货币

```
#include <iostream>
#include <locale>
#include <string>
#include <sstream>

using namespace std;

long double readMoney(istream& in, bool intl = false) {

   long double val;

   // Create a reader facet
   const money_get<char>& moneyReader =
     use_facet<money_get<char> >(in.getloc());

   // End marker
   istreambuf_iterator<char> end;

   // State variable for detecting errors
   ios_base::iostate state = 0;

   moneyReader.get(in, end, intl, in, state, val);

   // failbit will be set if something went wrong
   if (state != 0 && !(state & ios_base::eofbit))
      throw "Couldn't read money!\n";

   return(val);
}

void writeMoney(ostream& out, long double val, bool intl = false) {

   // Create a writer facet
   const money_put<char>& moneyWriter =
     use_facet<money_put<char> >(out.getloc());

   // Write to the stream. Call failed() (the return value is an
   // ostreambuf_iterator) to see if anything went wrong.
   if (moneyWriter.put(out, intl, out, out.fill(), val).failed())
      throw "Couldn't write money!\n";
}

int main() {

   long double val = 0;
   float exchangeRate = 0.775434f;  // Dollars to Euros
   locale locEn("english");
   locale locFr("french");

   cout << "Dollars: ";
   cin.imbue(locEn);
   val = readMoney(cin, false);

   cout.imbue(locFr);
   // Set the showbase flag so the currency char is printed
   cout.setf(ios_base::showbase);
   cout << "Euros: ";
   writeMoney(cout, val * exchangeRate, true);
}
```

如果你运行 13-6，你的输出如下所示：

```
Dollars: $100
Euros: EUR77,54
```

www.TopSage.com

讨论

这个money_put和money_get facet分别向一个流写入格式化的货币值或从从一个流中读货币值。它们和前几节中描述的date/time和数字的facet工作方式相同。标准需要提供一个窄字符或宽字符的示例，如money_put<char>和money_put<wchar_t>。就像其他的facet一样，这个get和put函数也是冗长的，但是一旦你多使用几次以后，它们的参数也是容易记住的。money_get和money_put都是使用一个moneypunct的类来存储格式化的信息。

首先，让我们来讨论向一个流写入货币值。货币值的显示涉及到几个方面：货币符号、正或者负的符号、千位分隔符和小数点。除了小数点之外的这些都是可选的。

你可以使用一个字符类型和一个locale来创建一个money_put，如下所示：

```
const money_put<char>& moneyWriter =
  use_facet<money_put<char> >(out.getloc());
```

money_put的char和wchar_t版本都是C++标准要求的。当你向一个流写入时使用与这个流相同的locale是一个好的主意，这样可以避免这个流和这个money_put写入的对象不匹配。接下来，调用put方法来把货币值写入输出流：

```
if (moneyWriter.put(out,        // Output iterator
                    intl,       // bool: use intl format?
                    out,        // ostream&
                    out.fill(), // fill char to use
                    val)        // currency value as long double
                    .failed())
    throw "Couldn't write money!\n";
```

money_put::put把日期写入输出流，这个输出流是你使用这个locale传递的，并且这个money_put对象也是你通过这个locale创建的。money_put::put返回一个指向最后一个字符输出之后的ostreambuf_iterator，这个ostreambuf_iterator有一个成员函数，并且你可以使用这个成员函数来检测这个迭代器是否处在一个被损坏的状态中。

money_put::put的参数都是自解释的，或许除了第二个之外（在这个例子中是这个intl参数）。它是一个布尔型值，用来确定是否使用货币符号（如$）还是使用国际化的三字符代码（如USD、EUR）。设置为false则是使用货币符号，设置成true则是使用国际化的三字符代码。

把货币输出到一个输出流中须遵循这个流上的一些格式标志。下面就是每个标志和它作用在货币上的效果：

ios_base::internal

在这个货币格式中无论是有空格还是没有任何东西，填充字符都会被使用（不是空格字符）。更多的关于使用格式的模式信息请参考下面有关moneypunct的讨论。

www.TopSage.com

ios_base::left 和 ios_base::right

导致货币值左对齐或右对齐，对齐过程中达到宽度值的余下空格用填充字符填充（参考下面的宽度描述）。这是方便的，因为它使得用表格格式化货币变得容易。

ios_base::width

money_put 值为流中字段宽度遵循标准规则。默认的情况下，值是左对齐的。如果字段比这个值大，money_put 的填充字符就会使用。

ios_base::showbase

当这个值为 true 时，货币符号就会打印出来；否则，不打印。

就像我在前面说的那样，money_get 和 money_put 都是用一个 moneypunct 的类，事实上是它来存储格式信息。你不需要考虑这个 moneypunct 类，除非你正在实现一个标准库，但是你可以用它来为某个特殊的 locale 探查格式信息。moneypunct 类包含的信息有使用的货币符号、为小数点使用的字符、正值和负值格式等等。示例 13-7 中为某个给定的 locale 呈现了一个打印货币格式信息的简短程序。

示例 13-7 打印货币信息

```
#include <iostream>
#include <locale>
#include <string>

using namespace std;

string printPattern(moneypunct<char>::pattern& pat) {

   string s(pat.field);  // pat.field is a char[4]
   string r;

   for (int i = 0; i < 4; ++i) {
      switch (s[i]) {
      case moneypunct<char>::sign:
        r += "sign ";
        break;
      case moneypunct<char>::none:
        r += "none ";
        break;
      case moneypunct<char>::space:
        r += "space ";
        break;
      case moneypunct<char>::value:
        r += "value ";
        break;
      case moneypunct<char>::symbol:
        r += "symbol ";
        break;
      }
   }
   return(r);
}

int main() {

   locale loc("danish");
```

www.TopSage.com

```
    const moneypunct<char>& punct =
      use_facet<moneypunct<char> >(loc);

    cout << "Decimal point:       " << punct.decimal_point() << '\n'
         << "Thousands separator: " << punct.thousands_sep() << '\n'
         << "Currency symbol:     " << punct.curr_symbol() << '\n'
         << "Positive sign:       " << punct.positive_sign() << '\n'
         << "Negative sign:       " << punct.negative_sign() << '\n'
         << "Fractional digits:   " << punct.frac_digits() << '\n'
         << "Positive format:"
         << printPattern(punct.pos_format()) << '\n'
         << "Negative format:"
         << printPattern(punct.neg_format()) << '\n';

    // Grouping is represented by a string of chars, but the meaning
    // of each char is its integer value, not the char it represents.
    string s = punct.grouping();
    for (string::iterator p = s.begin(); p != s.end(); ++p)
      cout << "Groups of: " << (int)*p << '\n';
}
```

这些方法的大部分都是自说明的，但是还是有一些需要进一步的说明。首先，这个grouping方法返回一个字符串，这个字符串解释为一串整型数。每个字符代表一个整型数对应的特殊索引的组，这个组从这个数的右边开始。如果对应这个索引没有值的话，就使用紧挨着这个索引的前面那个索引对应的值。换句话说，对于标准的美国格式，在这个字符串中，索引0处具有的值为3，也就意味着在索引0处，这些数字应该按照三个一组这样来分组。由于没有更多的值，也就意味着所有比0大的索引也应该是使用三个一组来分组。

pos_format和neg_format返回一个类型为moneypunct<T>::pattern的对象，它有一个T[4]的成员字段，这个T是一个字符类型。这个字段中的每一个元素都是枚举类型moneypunct<T>::part当中的一个枚举值，这个枚举类型有五个可能的值：none、space、symbol、sign和value。一个代表货币的字符串有四部分（因此这个数组的长度是四）。典型地，这些部分是货币符号、空格、正负值符号、值，如$ -32.00。常常，正值符号是一个空串，因为一个没有符号的值总是被认为是正的。负值符号可以多于一个字符，如“()”，在这种情况下，第一个字符打印在这个货币符号在neg_format格式中出现的位置，并且其余的打印在后面，因此你可以认为$(32.00)是一个负值。

大部分时间你不需要担心这个存储在moneypunct中的格式信息。但是你如果需要在不同的locale中处理很多货币格式的话，那么做一些试验来验证不同的locale中是如何格式化的就会很有价值。

请参考13.2节和13.3节。

13.5 给本地化字符串排序

问题

有一序列，它含有非ASCII字符的字符串，如何根据本地环境来对它们进行排序？

www.TopSage.com

大家网 TopSage.com

解决方案

这个locale类通过重载一个操作符为在一个给定的locale内比较字符提供了内在的支持。当你调用任何带有算符作为比较时的标准函数时，你可以使用这个locale类的实例作为你的比较算符（请参考示例13-8）。

示例13-8 特定于locale的排序

```
#include <iostream>
#include <locale>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

bool localeLessThan (const string& s1, const string& s2) {

   const collate<char>& col =
     use_facet<collate<char> >(locale()); // Use the global locale

   const char* pb1 = s1.data();
   const char* pb2 = s2.data();

   return (col.compare(pb1, pb1 + s1.size(),
                       pb2, pb2 + s2.size()) < 0);
}

int main() {

   // Create two strings, one with a German character
   string s1 = "di   ";
   string s2 = "dich";

   vector<string> v;
   v.push_back(s1);
   v.push_back(s2);

   // Sort without giving a locale, which will sort according to the
   // current global locale's rules.
   sort(v.begin(), v.end());
   for (vector<string>::const_iterator p = v.begin();
        p != v.end(); ++p)
      cout << *p << endl;

   // Set the global locale to German, and then sort
   locale::global(locale("german"));
   sort(v.begin(), v.end(), localeLessThan);
   for (vector<string>::const_iterator p = v.begin();
        p != v.end(); ++p)
      cout << *p << endl;
}
```

第一个排序遵循了ASCII排序的惯例，因此它的输出如下：

```
dich
diät
```

第二个排序根据德文的语义使用了正确的顺序，因此它的输出如下：

```
diät
dich
```

www.TopSage.com

讨论

当你在不同的环境中工作时，排序变得非常复杂，并且标准类库解决了这个问题。facet collate提供了一个成员函数compare，它就像strcmp函数：当第一个字符串小于第二字符串时，它返回-1，相等时返回0，而当第一个字符串大于第二个字符串时，返回1。与strcmp函数不同的是，collate::compare使用了目标locale中的字符语义。

示例13-8中说明了函数localeLessThan，根据这个全局的locale，如果第一个参数小于第二个参数，它就返回真值。这个函数中最重要的部分是调用compare函数。

```
col.compare(pb1,              // Pointer to the first char
            pb1 + s1.size(), // Pointer to one past the last char
            pb2,
            pb2 + s2.size())
```

依赖于你的实现中执行的字符集，示例13-8可能返回我前面说明的结果，也可能与之不同。但是如果你需要保证这个字符串的比较与某个特定于locale的方式相同，你就应该调用collate::compare。当然，标准没有要求实现支持除了“C”之外的任何locale，因此你需要通过测试来验证你的实现是否支持你需要支持的locale。

www.TopSage.com

第14章

XML

14.0 概述

XML在很多领域都是非常重要的，如信息存储和获取、发布和网络通信等等；在这一章中，你将学会如何在C++中处理XML。因为这本书是关于C++而不是XML的，所以我假定你对我讨论的一些XML相关的技术都有一定的经验，如SAX、DOM、XML schema、XPath和XSLT等。如果你并不是对所有的这些XML相关领域都熟悉，也不用着急；这一章的各节或多或少是相互独立的，因此你可以跳过某几节也能明白其余的。并且每一节都给出了它使用的XML的一些概念和工具的简单解释。

如果你熟悉另外的一些编程语言，如JAVA，你或许期望在C++的标准类库中找到一些处理XML的一些工具。不幸的是，当C++标准形成时，XML还处在它的发展初期，或许在未来的C++标准类库版本中将会增加XML处理的一些库，但现在在C++中你不得不依赖于那些第三方的XML类库。

在你开始阅读这一章的各节之前，你或许想下载和安装本章中我要讨论的那些类库。表14-1给出了每个类库的主页；表14-2说明了每个库的一些功能和本章讨论这些类库的章节。这两个表没有给出每个库对应的XML规范的级别，因为这些库也是在不断地改变当中。

表14-1 C++中处理XML的类库

库名	主页
TinyXml	*www.grinninglizard.com/tinyxml*
Xerxes	*xml.apache.org/xerces-c*
Xalan	*xml.apache.org/xalan-c*
Pathan 1	*software.decisionsoft.com/pathanIntro.html*
Boost.Serialization	*www.boost.org/libs/serialization*

www.TopSage.com

表14-2 这些类库是如何使用的

库名	功能特征	章节
TinyXml	DOM（非标准）	14.1
Xerxes	SAX2、DOM、XML Schema	14.2～14.8
Xalan	XSLT、XPath	14.7～14.8
Pathan	XPath	14.8
Boost.Serialization	XML Serialization	14.9

14.1 解析一个简单的XML文档

问题

有一个存储在XML文档中的数据集合，如何解析这个文档，并把这些数据存到一个C++集合对象中？你的XML文档足够小以至于它可以全部放到内存中，并且不需要使用内部文档类型定义（DTD）或者XML命名空间。

解决方案

使用TinyXml类库。首先，定义一个类型为`TiXmlDocument`的对象并且调用它的`LoadFile()`方法，把你的XML文档的路径名作为参数传给它。如果`LoadFile()`方法返回真值的话，就表明你的XML文档被成功地解析。如果解析成功，就调用`RootElement()`方法来获得这个类型为`TiXmlElement`的对象，它代表这个XML文档的根对象。这个对象具有一个层次结构，它反映了你的XML文档的结构；通过遍历这个结构，你就可以获取关于你的文档的信息并且使用这些信息来创建一个C++的集合对象。

例如，假设你有一个XML文档*animals.xml*，它代表马戏团动物的集合，如示例14-1所示。这个文档的根是一个名为`animalList`的元素并且它还有很多子动物元素，每一个都代表Feldman家庭马戏团拥有的一个动物。假设你有一个名为Animal的C++类，并且你想根据这个文档中相应的动物来构造一个`Animal`类型的`std::vector`。

示例14-1 一个表示马戏团动物的XML文档

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Feldman Family Circus Animals -->

<animalList>
    <animal>
        <name>Herby</name>
        <species>elephant</species>
        <dateOfBirth>1992-04-23</dateOfBirth>
        <veterinarian name="Dr. Hal Brown" phone="(801)595-9627"/>
        <trainer name="Bob Fisk" phone="(801)881-2260"/>
    </animal>
```

www.TopSage.com

```
    <animal>
        <name>Sheldon</name>
        <species>parrot</species>
        <dateOfBirth>1998-09-30</dateOfBirth>
        <veterinarian name="Dr. Kevin Wilson" phone="(801)466-6498"/>
        <trainer name="Eli Wendel" phone="(801)929-2506"/>
    </animal>
    <animal>
        <name>Dippy</name>
        <species>penguin</species>
        <dateOfBirth>2001-06-08</dateOfBirth>
        <veterinarian name="Dr. Barbara Swayne" phone="(801)459-7746"/>
        <trainer name="Ben Waxman" phone="(801)882-3549"/>
    </animal>
</animalList>
```

示例14-2说明了这个Animal类的定义。Animal有五个数据成员，这五个数据成员对应于这个动物的名字、种类、出生日期、兽医和培训师。其中动物的名字和种类是std::string类型，出生日期是boost::gregorian::date类型，这个类型来自于Boost.Date_Time类型，并且它的兽医和培训师则是Contact类的一个实例，这个Contact类定义在示例14-2中。示例14-3说明如何使用TinyXml来解析这个*animals.xml*文档，遍历这个解析好的文档并且使用从这个文档中获取到的数据来填充这个Animals类型的std::vector。

示例 14-2 animal.hpp 头文件

```
#ifndef ANIMALS_HPP_INCLUDED
#define ANIMALS_HPP_INCLUDED

#include <ostream>
#include <string>
#include <stdexcept> // runtime_error
#include <boost/date_time/gregorian/gregorian.hpp>
#include <boost/regex.hpp>

// Represents a veterinarian or trainer
class Contact {
public:
    Contact() { }
    Contact(const std::string& name, const std::string& phone)
        : name_(name)
    {
        setPhone(phone);
    }
    std::string name() const { return name_; }
    std::string phone() const { return phone_; }
    void setName(const std::string& name) { name_ = name; }
    void setPhone(const std::string& phone)
    {
        using namespace std;
        using namespace boost;
        // Use Boost.Regex to verify that phone
        // has the form (ddd)ddd-dddd
        static regex pattern("\\([0-9]{3}\\)[0-9]{3}-[0-9]{4}");
        if (!regex_match(phone, pattern)) {
            throw runtime_error(string("bad phone number:") + phone);
        }
        phone_ = phone;
    }
private:
    std::string name_;
```

www.TopSage.com

大家网 TopSage.com

```
    std::string phone_;
};

// Compare two Contacts for equality; used in Recipe 14.9
// (for completeness, you should also define operator!=)
bool operator==(const Contact& lhs, const Contact& rhs)
{
    return lhs.name() == rhs.name() && lhs.phone() == rhs.phone();
}

// Writes a Contact to an ostream
std::ostream& operator<<(std::ostream& out, const Contact& contact)
{
    out << contact.name() << " " << contact.phone();
    return out;
}

// Represents an animal
class Animal {
public:
    // Default constructs an Animal; this is
    // the constructor you'll use most
    Animal() { }

    // Constructs an Animal with the given properties;
    // you'll use this constructor in Recipe 14.9
    Animal( const std::string& name,
            const std::string& species,
            const std::string& dob,
            const Contact& vet,
            const Contact& trainer )
        : name_(name),
          species_(species),
          vet_(vet),
          trainer_(trainer)
    {
        setDateOfBirth(dob);
    }

    // Getters
    std::string             name() const { return name_; }
    std::string             species() const { return species_; }
    boost::gregorian::date  dateOfBirth() const { return dob_; }
    Contact                 veterinarian() const { return vet_; }
    Contact                 trainer() const { return trainer_; }

    // Setters
    void setName(const std::string& name) { name_ = name; }
    void setSpecies(const std::string& species) { species_ = species; }
    void setDateOfBirth(const std::string& dob)
    {
        dob_ = boost::gregorian::from_string(dob);
    }
    void setVeterinarian(const Contact& vet) { vet_ = vet; }
    void setTrainer(const Contact& trainer) { trainer_ = trainer; }
private:
    std::string             name_;
    std::string             species_;
    boost::gregorian::date  dob_;
    Contact                 vet_;
    Contact                 trainer_;
};

// Compare two Animals for equality; used in Recipe 14.9
// (for completeness, you should also define operator!=)
bool operator==(const Animal& lhs, const Animal& rhs)
{
```

www.TopSage.com

```
    return lhs.name() == rhs.name() &&
           lhs.species() == rhs.species() &&
           lhs.dateOfBirth() == rhs.dateOfBirth() &&
           lhs.veterinarian() == rhs.veterinarian() &&
           lhs.trainer() == rhs.trainer();
}

// Writes an Animal to an ostream
std::ostream& operator<<(std::ostream& out, const Animal& animal)
{
    out << "Animal {\n"
        << "  name=" << animal.name() << ";\n"
        << "  species=" << animal.species() << ";\n"
        << "  date-of-birth=" << animal.dateOfBirth() << ";\n"
        << "  veterinarian=" << animal.veterinarian() << ";\n"
        << "  trainer=" << animal.trainer() << ";\n"
        << "}";
    return out;
}

#endif // #ifndef ANIMALS_HPP_INCLUDED
```

示例 14-3 使用 TinyXml 解析 animals.xml

```
#include <exception>
#include <iostream>     // cout
#include <stdexcept>    // runtime_error
#include <cstdlib>      // EXIT_FAILURE
#include <cstring>      // strcmp
#include <vector>
#include <tinyxml.h>
#include "animal.hpp"

using namespace std;

// Extracts the content of an XML element that contains only text
const char* textValue(TiXmlElement* e)
{
    TiXmlNode* first = e->FirstChild();
    if ( first != 0 &&
         first == e->LastChild() &&
         first->Type() == TiXmlNode::TEXT )
    {
        // the element e has a single child, of type TEXT;
        // return the child's
        return first->Value();
    } else {
        throw runtime_error(string("bad ") + e->Value() + " element");
    }
}

// Constructs a Contact from a "veterinarian" or "trainer" element
Contact nodeToContact(TiXmlElement* contact)
{
    using namespace std;
    const char *name, *phone;
    if ( contact->FirstChild() == 0 &&
         (name = contact->Attribute("name")) &&
         (phone = contact->Attribute("phone")) )
    {
        // The element contact is childless and has "name"
        // and "phone" attributes; use these values to
        // construct a Contact
        return Contact(name, phone);
    } else {
        throw runtime_error(string("bad ") + contact->Value() + " element");
    }
}
```

www.TopSage.com

```
// Constructs an Animal from an "animal" element
Animal nodeToAnimal(TiXmlElement* animal)
{
    using namespace std;

    // Verify that animal corresponds to an "animal" element
    if (strcmp(animal->Value(), "animal") != 0) {
        throw runtime_error(string("bad animal: ") + animal ->Value());
    }

    Animal result; // Return value
    TiXmlElement* element = animal->FirstChildElement();

    // Read name
    if (element && strcmp(element->Value(), "name") == 0) {
        // The first child element of animal is a "name"
        // element; use its text value to set the name of result
        result.setName(textValue(element));
    } else {
        throw runtime_error("no name attribute");
    }

    // Read species
    element = element->NextSiblingElement();
    if (element && strcmp(element->Value(), "species") == 0) {
        // The second child element of animal is a "species"
        // element; use its text value to set the species of result
        result.setSpecies(textValue(element));
    } else {
        throw runtime_error("no species attribute");
    }

    // Read date of birth
    element = element->NextSiblingElement();
    if (element && strcmp(element->Value(), "dateOfBirth") == 0) {
        // The third child element of animal is a "dateOfBirth"
        // element; use its text value to set the date of birth
        // of result
        result.setDateOfBirth(textValue(element));
    } else {
        throw runtime_error("no dateOfBirth attribute");
    }

    // Read veterinarian
    element = element->NextSiblingElement();
    if (strcmp(element->Value(), "veterinarian") == 0) {
        // The fourth child element of animal is a "veterinarian"
        // element; use it to construct a Contact object and
        // set result's veterinarian
        result.setVeterinarian(nodeToContact(element));
    } else {
        throw runtime_error("no veterinarian attribute");
    }

    // Read trainer
    element = element->NextSiblingElement();
    if (strcmp(element->Value(), "trainer") == 0) {
        // The fifth child element of animal is a "trainer"
        // element; use it to construct a Contact object and
        // set result's trainer
        result.setTrainer(nodeToContact(element));
    } else {
        throw runtime_error("no trainer attribute");
    }

    // Check that there are no more children
    element = element->NextSiblingElement();
    if (element != 0) {
```

www.TopSage.com

```
        throw runtime_error(
                  string("unexpected element:") +
                  element->Value()
              );

    }

    return result;
}

int main()
{
    using namespace std;

    try {
        vector<Animal> animalList;

        // Parse "animals.xml"
        TiXmlDocument doc("animals.xml");
        if (!doc.LoadFile())
            throw runtime_error("bad parse");

        // Verify that root is an animal-list
        TiXmlElement* root = doc.RootElement();
        if (strcmp(root->Value(), "animalList") != 0) {
            throw runtime_error(string("bad root: ") + root->Value());
        }

        // Traverse children of root, populating the list
        // of animals
        for ( TiXmlElement* animal = root->FirstChildElement();
              animal;
              animal = animal->NextSiblingElement() )
        {
              animalList.push_back(nodeToAnimal(animal));
        }

        // Print the animals' names
        for ( vector<Animal>::size_type i = 0,
                                        n = animalList.size();
              i < n;
              ++i )
        {
              cout << animalList[i] << "\n";
        }
    } catch (const exception& e) {
        cout << e.what() << "\n";
        return EXIT_FAILURE;
    }
}
```

讨论

对于这种简单的XML处理程序，TinyXml是一个最好的选择。它的源代码很少，并且很容易和项目整合，接口也很简单。同时它的许可证也非常适合。它的主要缺点就是它不支持XML命名空间，不能使用DTD或者schema来对XML文档做合法性检查，并且不能处理内部包含DTD的XML文档。如果你需要这些功能特性或者其他的XML相关的技术（如XPath或XSLT）的话，你需要使用本章描述的其他类库。

这个TinyXml解析器产生一个代表XML文档的树，这个树的节点代表XML文档的元素、文本、注释和其他的XML文档组件。这个树的根代表XML文档本身。这个作为树的层

www.TopSage.com

次文档就叫做文档对象模型（DOM）。这个 TinyXml 的 DOM 和 W3C 联盟设计的 DOM 类似，但它不遵从W3C的规范。在保持TinyXml的最小特性下，TinyXml的DOM比W3C简单，但也没有 W3C 的 DOM 功能强大。

这个代表XML文档的树的节点可以通过TiXmlNode接口来访问，这个接口提供了方法来访问节点的父节点、枚举所有的子节点、删除子节点或者插入一些附加的子节点。每个节点事实上都是某个继承的类型的实例；例如，这个树的根节点是TiXmlDocument的一个实例，代表元素的节点是TiXmlElement 的实例，而代表文本的节点是TiXmlText 的一个实例。TiXmlNode 节点的类型可以通过调用 Type() 方法来获取；一旦你知道一个节点的类型，你就可以调用某些方便的方法如 toDocument()、toElement() 和 toText() 来获取它所代表的内容。这些继承的类型还包含一些与这个继承类型相对应的方法。

示例 14-3 是很容易明白的。首先，这个 textValue() 函数可以获取这个仅包含文本的元素的文本内容，如名字、种类或者出生日期等。它首先验证这个元素只有一个子节点并且这个子节点是一个文本节点。然后通过调用Value() 方法来获取这个子节点的文本内容，这个方法可以返回这个文本节点或注释节点的内容、元素节点的标签名字和一个根节点的文件名字。

接下来nodeToContact() 函数带一个对应于兽医或培训师元素的节点并且从它的名字和电话属性的值构造一个Contact对象，而这些属性可以通过使用Attribute() 方法来获取。

相同地，这个 nodeToAnimal() 带一个对应于动物元素的节点并且构造一个 Animal 对象。它这样做是通过使用 NextSiblingElement() 方法来枚举它的每个子节点，然后获取包含在每一个元素中的数据，并且给这个 Animal 对象设置相应的属性。这些数据的获取是通过在元素的 name、species 和 dateOfBirth 上调用 textValue() 方法得到的，并且通过在兽医和培训师元素上使用 nodeToContact() 获得对应的数据。

在这个主函数中，我首先构造一个对应于文件 *animals.xml* 的 TiXmlDocument 对象并且使用 LoadFile() 方法来解析它。然后我调用 RootElement() 方法来获取对应于这个 XML 文档根节点的 TiXmlElement 对象。接下来，我再枚举这个根节点元素的所有子节点，使用nodeToAnimal() 来给每一个动物元素构造一个 Animal 对象。最后，我再枚举每一个 Animal 对象并把它们输出到标准输出上。

在示例 14-3 中没有谈到的关于 TinyXml 类库的一个特性就是 TiXmlDocument 类的 SaveFile() 方法，它把这个代表 XML 文档的 TiXmlDocument 写到一个文件中。这就可以让你解析一个 XML 文档，使用 DOM 接口修改它，并且把这个修改过的文档保存到一个文件中。你甚至可以从一个草稿中创建一个 TiXmlDocument 并把它保存到硬盘中：

www.TopSage.com

大家网 TopSage.com

```
// Create a document hello.xml, consisting
// of a single "hello" element
TiXmlDocument doc;
TiXmlElement root("hello");
doc.InsertEndChild(root);
doc.SaveFile("hello.xml");
```

请参考 14.3 节和 14.4 节。

14.2 使用 Xerces 字符串

问题

如何安全地和容易地使用 Xerces 类库来处理宽字符串？特别地，你需要能够存储由 Xerces 函数返回的字符串，并且在 Xerces 字符串和 C++ 标准类库字符串之间相互转换。

解决方案

你可以使用由 Xerces 宽字符类型 XMLCh 特殊化模板 `std::basic_string` 的 Xerces 类库函数来存储宽字符串。

```
typedef std::basic_string<XMLCh> XercesString;
```

为了在 Xerces 字符串和窄字符串之间相互转换，可以使用来自类 `xercesc::XMLString` 的重载静态方法 `transcode()`，这个方法定义在头文件 *xercesc/util/XMLString.hpp* 中。示例 14-4 定义了两个重载函数 `toNative` 和 `fromNative` 函数，它们使用 `transcode` 函数来在窄字符串和 Xerces 字符串之间相互转换。每个函数都有两个变体，一个带有 C 风格的字符串，而另一个带有 C++ 风格的字符串。你可以使用这些函数在 Xerces 字符串和窄字符串之间相互转换；一旦定义了它们，你就可以不用再直接调用 `transcode` 方法。

示例 14-4 头文件 xerces_strings.hpp，它用于在 Xerces 字符串和窄字符串之间相互转换

```
#ifndef XERCES_STRINGS_HPP_INCLUDED
#define XERCES_STRINGS_HPP_INCLUDED

#include <string>
#include <boost/scoped_array.hpp>
#include <xercesc/util/XMLString.hpp>

typedef std::basic_string<XMLCh> XercesString;

// Converts from a narrow-character string to a wide-character string.
inline XercesString fromNative(const char* str)
{
    boost::scoped_array<XMLCh> ptr(xercesc::XMLString::transcode(str));
    return XercesString(ptr.get());
}

// Converts from a narrow-character string to a wide-charactr string.
inline XercesString fromNative(const std::string& str)
{
    return fromNative(str.c_str());
}

// Converts from a wide-character string to a narrow-character string.
inline std::string toNative(const XMLCh* str)
```

www.TopSage.com

```
{
    boost::scoped_array<char> ptr(xercesc::XMLString::transcode(str));
    return std::string(ptr.get());
}

// Converts from a wide-character string to a narrow-character string.
inline std::string toNative(const XercesString& str)
{
    return toNative(str.c_str());
}

#endif // #ifndef XERCES_STRINGS_HPP_INCLUDED
```

为了在Xerces字符串和std::wstring字符串之间相互转换，简单地使用std::basic_string构造函数，它带一对迭代器作为参数。例如，你可以如下这样定义两个函数：

```
// Converts from a Xerces String to a std::wstring
std::wstring xercesToWstring(const XercesString& str)
{
    return std::wstring(str.begin(), str.end());
}

// Converts from a std::wstring to a XercesString
XercesString wstringToXerces(const std::wstring& str)
{
    return XercesString(str.begin(), str.end());
}
```

这些函数都依赖于一个事实，那就是wchar_t和XMLCh都是完整的类型，它们都可以隐式地相互之间转换；并且不用考虑这个wchar_t的宽度大小，只要它的值没有超过XMLCh使用的范围就可以。你也可以使用带一个字符数组和它的长度作为参数的std::basic_string构造函数来定义带C风格字符串作为参数的类似函数。

讨论

Xerces使用以null结尾的XMLCh字符类型序列来代表Unicode字符串。XMLCh是一个实现定义好的完整类型的typedef，这个完整类型具有最少16位的字符宽度，这样足够代表现在在任何语言当中使用的已知字符。Xerces使用UTF-16字符编码，理论上就意味着一些Unicode字符必须通过多个XMLCh字符的序列来代表；然而实际上，你可以把一个XMLCh字符直接想象成一个Unicode编码点，例如一个Unicode字符的数字值。

曾经，XMLCh被定义成一个XMLCh的typedef，这就意味着你可以很容易地作为一个std::wstring来存储一个Xerces字符。然而同时，在所有的平台上Xerces类库又把XMLCh重定义成一个无符号短整型。也就意味着在一个平台上XMLCh和wchar_t不具有相同的宽度。由于Xerces可能在将来改变这个XMLCh的定义，所以你不能指望XMLCh与任何特殊的类型都相同。因此如果你需要存储一个Xerces字符串的副本，你应该使用std::basic_string<XMLCh>。

当使用Xerces类库时，你需要经常在窄字符串和Xerces字符串之间相互转换；为了这个目的，Xerces提供了一个重载函数transcode()。transcode()函数可以把一个

www.TopSage.com

Unicode字符串转换成一个使用本地字符编码的窄字符串或者把一个使用本地字符编码的窄字符串转换成一个Unicode字符串。这个本地编码并没有精确的定义，然而，如果你在一个有多个经常使用的字符编码的环境中编程时，你就需要考虑你自己的处理并执行一些你自己的转换，或者通过使用std::codecvt功能，或者使用支持Xerces可插拔的代码转换服务，这个服务在Xerces的文档中有相应的描述。在很多情况下，transcode()可以满足你的需求。

这个由transcode()返回的一个null结尾的字符串是使用new操作符分配的一个数组形式，因此你需要使用delete[]来删除它。这导致了一个轻微的内存管理问题，由于典型地你需要在删除这个字符串之前复制它或把它输出到一个流中去，而这些操作是有可能抛出异常的。在示例14-4中我通过使用模板boost::scoped_array来阐述这个问题，这个模板可以动态地分配一个数组并且当它超出了它的生存空间时自动地删除它，即使是在抛出异常的情况下也能删除它。例如，看一下fromNative函数的实现：

```
inline XercesString fromNative(const char* str)
{
    boost::scoped_array<XMLCh> ptr(xercesc::XMLString::transcode(str));
    return XercesString(ptr.get());
}
```

这里，ptr拥有这个由transcode()返回的以null结尾的字符串，并且即使是XercesString构造函数抛出一个std::bad_alloc异常的情况下也能释放它。

14.3 解析一个复杂的XML文档

问题

有一序列的数据存储在一个使用内部DTD或者XML命名空间的XML文档中。如何解析这个文档，并且把这些数据存储到一个C++的集合对象中？

解决方案

使用SAX2 API（XML简单API，2.0版本）的Xerces实现。首先，从xercesc::ContentHandler派生出一个类；这个类将收到XML文档中结构和内容被解析的通知。接下来，如果你愿意，再从xercesc::ErrorHandler派生出一个类来接收警告和错误通知。构造一个xercesc::SAX2XMLReader类型的解析器，使用这个解析器的setContentHandler()和setErrorHandler()方法注册一个handler类的实例。最后调用解析器的parse()方法，给这个方法传入你的文档文件的路径名作为参数。

例如，假设你想解析示例14-1中的*animals.xml*的XML文档，并且构造一个Animal类型的std::vector来代表这个文档中的动物（有关Animal类的定义，请参考示例14-2）。示例14-3中，我说明了如何使用TinyXml。为了使得这个问题更具有挑战性，让我们给这个文档加上命名空间，如示例14-5所示。

www.TopSage.com

示例 14-5 使用 XML 命名空间，列举马戏团的动物

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Feldman Family Circus Animals with Namespaces -->

<ffc:animalList xmlns:ffc="http://www.feldman-family-circus.com">
    <ffc:animal>
        <ffc:name>Herby</ffc:name>
        <ffc:species>elephant</ffc:species>
        <ffc:dateOfBirth>1992-04-23</ffc:dateOfBirth>
        <ffc:veterinarian name="Dr. Hal Brown" phone="(801)595-9627"/>
        <ffc:trainer name="Bob Fisk" phone="(801)881-2260"/>
    </ffc:animal>

    <!-- etc. -->

</ffc:animalList>
```

为了使用 SAX2 来解析这个文档，定义一个 ContentHandler，如示例 14-6 所示，并且定义一个 ErrorHandler，如示例 14-7 所示。然后构造一个 SAX2XMLReader，并注册你自己的处理器，然后运行这个解析器。如示例 14-8 所示。

示例 14-6 解析 animals.xml 文档的 SAX2 ContentHandler

```
#include <stdexcept>                          // runtime_error
#include <vector>
#include <xercesc/sax2/Attributes.hpp>
#include <xercesc/sax2/DefaultHandler.hpp> // Contains no-op
                                           // implementations of
                                           // the various handlers
#include "xerces_strings.hpp"                 // Example 14-4
#include "animal.hpp"

using namespace std;
using namespace xercesc;

// Returns an instance of Contact based
// on the given collection of attributes
Contact contactFromAttributes(const Attributes &attrs)
{
    // For efficiency, store frequently used string
    // in static variables
    static XercesString name = fromNative("name");
    static XercesString phone = fromNative("phone");

    Contact result;    // Contact to be returned.
    const XMLCh* val;  // Value of name or phone attribute.

    // Set Contact's name.
    if ((val = attrs.getValue(name.c_str())) != 0) {
        result.setName(toNative(val));
    } else {
        throw runtime_error("contact missing name attribute");
    }

    // Set Contact's phone number.
    if ((val = attrs.getValue(phone.c_str())) != 0) {
        result.setPhone(toNative(val));
    } else {
        throw runtime_error("contact missing phone attribute");
    }

    return result;
}
```

www.TopSage.com

```
// Implements callbacks that receive character data and
// notifications about the beginnings and ends of elements
class CircusContentHandler : public DefaultHandler {
public:
    CircusContentHandler(vector<Animal>& animalList)
        : animalList_(animalList)
        { }

    // If the current element represents a veterinarian or trainer,
    // use attrs to construct a Contact object for the current
    // Animal; otherwise, clear currentText_ in preparation for the
    // characters() callback
    void startElement(
           const XMLCh *const uri,        // namespace URI
           const XMLCh *const localname,  // tagname w/ out NS prefix
           const XMLCh *const qname,      // tagname + NS pefix
           const Attributes &attrs )      // elements's attributes
    {
        static XercesString animalList = fromNative("animalList");
        static XercesString animal = fromNative("animal");
        static XercesString vet = fromNative("veterinarian");
        static XercesString trainer = fromNative("trainer");
        static XercesString xmlns =
            fromNative("http://www.feldman-family-circus.com");

        // Check namespace URI
        if (uri != xmlns)
            throw runtime_error(
                      string("wrong namespace uri: ") + toNative(uri)
                  );
        if (localname == animal) {
            // Add an Animal to the list; this is the new
            // "current Animal"
            animalList_.push_back(Animal());
        } else if (localname!= animalList) {
            Animal& animal = animalList_.back();
            if (localname == vet) {
                // We've encountered a "veterinarian" element.
                animal.setVeterinarian(contactFromAttributes(attrs));
            } else if (localname == trainer) {
                // We 've encountered a "trainer" element.
                animal.setTrainer(contactFromAttributes(attrs));
            } else {
                // We've encountered a "name" , "species", or
                // "dateOfBirth" element. Its content will be supplied
                // by the callback function characters().
                currentText_.clear();
             }
         }
    }

    // If the current element represents a name, species, or date
    // of birth, use the text stored in currentText_ to set the
    // appropriate property of the current Animal.
    void endElement(
           const XMLCh *const uri,        // namespace URI
           const XMLCh *const localname,  // tagname w/ out NS prefix
           const XMLCh *const qname )     // tagname + NS pefix
    {
        static XercesString animalList = fromNative("animal-list");
        static XercesString animal = fromNative("animal");
        static XercesString name = fromNative("name");
        static XercesString species = fromNative("species");
        static XercesString dob = fromNative("dateOfBirth");

        if (localname!= animal && localname!= animalList) {
            // currentText_ contains the content of the element
            // which has ended. Use it to set the current Animal's
```

www.TopSage.com

大家网 TopSage.com

```
            // properties.
            Animal& animal = animalList_.back();
            if (localname == name) {
                animal.setName(toNative(currentText_));
        } else if (localname == species) {
            animal.setSpecies(toNative(currentText_));
        } else if (localname == dob) {
            animal.setDateOfBirth(toNative(currentText_));
        }
      }
    }
    // Receives notifications when character data is encountered
    void characters( const XMLCh* const chars,
                     const unsigned int length )
    {
        // Append characters to currentText_ for processing by
        // the method endElement()
        currentText_.append(chars, length);
    }
private:
    vector<Animal>&  animalList_;
    XercesStringcurrentText_;
};
```

示例 14-7 SAX2 ErrorHandler 错误处理器

```
#include <stdexcept> // runtime_error
#include <xercesc/sax2/DefaultHandler.hpp>

// Receives Error notifications.
class CircusErrorHandler : public DefaultHandler {
public:
    void warning(const SAXParseException& e)
    {
        /* do nothing */
    }
    void error(const SAXParseException& e)
    {
        throw runtime_error(toNative(e.getMessage()));
    }
    void fatalError(const SAXParseException& e) { error(e); }
};
```

示例 14-8 使用 SAX2 API 来解析文档 animals.xml

```
#include <exception>
#include <iostream>// cout
#include <memory>  // auto_ptr
#include <vector>
#include <xercesc/sax2/SAX2XMLReader.hpp>
#include <xercesc/sax2/XMLReaderFactory.hpp>
#include <xercesc/util/PlatformUtils.hpp>
#include "animal.hpp"
#include "xerces_strings.hpp"  // Example 14-4

using namespace std;
using namespace xercesc;

// RAII utility that initializes the parser and frees resources
// when it goes out of scope
class XercesInitializer {
public:
    XercesInitializer() { XMLPlatformUtils::Initialize(); }
    ~XercesInitializer() { XMLPlatformUtils::Terminate(); }
private:
    // Prohibit copying and assignment
    XercesInitializer(const XercesInitializer&);
    XercesInitializer& operator=(const XercesInitializer&);
```

www.TopSage.com

```
};

int main()
{
    try {
        vector<Animal> animalList;

        // Initialze Xerces and obtain parser
        XercesInitializer   init;
        auto_ptr<SAX2XMLReader>  ]
            parser(XMLReaderFactory::createXMLReader());

        // Register handlers
        CircusContentHandlercontent(animalList);
        CircusErrorHandler  error;
        parser->setContentHandler(&content);
        parser->setErrorHandler(&error);

        // Parse the XML document
        parser->parse("animals.xml");

        // Print animals' names
        for ( vector<Animal>::size_type i = 0,
                                        n = animalList.size();
              i < n;
              ++i )
        {
            cout << animalList[i] << "\n";
        }
    } catch (const SAXException& e) {
        cout << "xml error: " << toNative(e.getMessage()) << "\n";
        return EXIT_FAILURE;
    } catch (const XMLException& e) {
        cout << "xml error: " << toNative(e.getMessage()) << "\n";
        return EXIT_FAILURE;
    } catch (const exception& e) {
        cout << e.what() << "\n";
        return EXIT_FAILURE;
    }
}
```

讨论

一些XML解析器解析XML文档并且返回给用户一个复杂的C++对象。接下来的几节中的TinyXml和W3C DOM解析器都是以这种方式工作。相反，SAX2解析器使用一序列的回调函数来传达XML文档的有关信息给用户。这些回调函数被组织成一个处理接口：ContentHandler接收关于XML文档的元素、属性和文本信息的通知，ErrorHandler接收警告和错误通知，而DTDHandler则接收有关这个XML文档的DTD的通知。

围绕着这一序列的回调函数来设计解析器有很多重要的优点。例如，使得解析那些太大而不能完全装入内存的文档成为可能。此外，它避免无数的为构造XML文档的内部节点的动态分配，并且能让用户直接构造这个文档中他自己的节点，而不是像我在示例14-3中那样在这整个节点树中贯穿，从而能够节省处理时间。

示例14-8是非常直接的：我获取一个SAX2解析器，注册一个ContentHandler和ErrorHandler，然后解析这个animals.xml文档，然后打印由ContentHandler获得的

www.TopSage.com

Animal类的列表。这里有两点比较有趣：首先，函数XMLReaderFactory::createXMLReader()返回一个动态分配的SAX2XMLReader实例，这个实例必须由用户显式地释放；为这个目的我使用std::auto_ptr来保证即使在有异常抛出的情况下这个实例也能被删除。第二，这个Xerces的框架必须使用xercesc::XMLPlatformUtils::Initialize()方法来初始化并且必须使用xercesc::XMLPlatformUtils::Terminate()方法来清除。我在XercesInitializer类中封装了这个初始化和清除动作，这个类在构造函数中调用XMLPlatformUtils::Initialize()方法，在它的析构函数中调用XMLPlatformUtils::Terminate()方法。这就能保证即使有异常抛出的情况下也能调用 Terminate()方法。这是示例 8-3 中说明的资源获取初始化（RAII）技术的一个例子。

让我们看看示例14-6中的CircusContentHandler类是如何实现SAX2 ContentHandler接口的。这个 SAX2 的解析器每当碰到一个元素的开始标签时就调用 startElement()方法。如果这个元素有一个相关的命名空间，这第一个参数 uri 就包含这个元素的命名空间 URI，并且第二个参数 localname 包含紧跟在命名空间前置的元素标签名的一部分。如果这个元素没有相关的命名空间，这两个参数就都是空的字符串。第三个参数包含这个元素的标签名；如果这个元素确实有相关的命名空间的话，这个参数可能包含这个元素的标签名，就好像它出现在正在被处理的文档中一样，但它也可能是一个空的字符串。第四个参数是类 Attribute 的一个实例，这个类代表这个元素的属性集。

在示例 14-6 中 startElement()方法的实现中，我忽略了 animalList 元素。当我碰到一个动物元素时，我就在这个动物列表中增加一个新的 Animal 对象，让我们把这个 Animal 对象当成是当前的 Animal 对象，并且把给这个 Animal 对象设置属性的工作交给处理器 handler。当我碰到一个 veterinarian 或 trainer 元素时，我就调用函数contactFromAttributes来从这个元素的属性集中构造一个Contact类的实例，然后使用这个 Contact 类的实例来设置这个 Animal 对象的 veterinarian 或 trainer 属性。当我碰到一个 name、species 或 dateOfBirth 元素时，我就清除这个成员变量 currentText_，这个成员变量用来存储这个元素的文本内容。

这个 SAX2 解析器调用 characters()方法来分发这个元素包含的字符数据。 这个解析器按照串行的方式调用characters()来分发元素的字符，直到碰到这个元素的结束标签为止，但是不能保证这个元素的所有字符数据都能被分发。因此，在这个 characters()方法的实现中，我简单地把提供的字符追加到这个成员变量 currentText_中，只要碰到 name、species 或者 dateofbirth 的结束标签时，这个成员变量就用来设置当前 Animal 对象的 name、species 或 dateofbirth。

每当 SAX2 解析器离开一个元素时就调用 endElement()方法。它的参数与 startElement()方法的前三个参数具有相同的阐述。在示例 14-6 中 endElement()方法的实现中，我忽略除了 name、species 和 dateOfBirth 的所有元素。当对应于这

www.TopSage.com

些元素中的某个元素的回调函数发生时，标志着这个解析器正离开这个元素，我使用存储在currentText_中的字符数据来设置当前的Animal对象的name、species或者dateofbirth。

SAX2的几个重要特性在示例14-6、14-7和14-8中都没有提到。例如，类SAX2XMLReader提供了parse()方法的一个重载函数，这个重载函数带一个xercesc::InputSource的实例作为参数来代替C语言风格的字符串。InputSource是一个封装了源字符数据的抽象类；它的具体子类包含xercesc::MemBufInputSource和xercesc::URLInputSource，让SAX2解析器来处理存储在任何位置上的XML文档，而不只是本地文件系统中的XML文档。

此外，这个ContentHandler包含很多附加的方法，如startDocument()和endDocmuent()，这标志着一个XML文档的开始和结束；以及setLocator()，它让你能指定一个Locator对象，这个Locator对象跟踪了当前正在处理的文件的当前位置。还有一些别的处理器接口，包含DTDHandler和EntityResolver，它们来自核心SAX2.0规范；以及DeclarationHandler和LexicalHandler，它们来自SAX2.0的标准扩展版本。

一个简单的类实现多个处理器接口也是可能的。类xercesc::DefaultHandler使得这项工作变得容易，因为它继承了所有的这些处理器接口并提供了这个虚函数的无操作的实现。因此，我把这个方法加到了从CircusErrorHandler到CircusContentHandler处理器中，示例14-8中的修改如下所示：

```
// Register handlers
CircusContentHandler handler(animalList);
parser->setContentHandler(&handler);
parser->setErrorHandler(&handler);
```

示例14-8中最后一个你应该注意的特性就是：CircusContentHandler并没有企图去验证这个正在解析的文档具有正确的结构，例如，它的根是一个animalList元素或者这个根的所有子元素都是animal元素。与示例14-3相反。例如，在示例14-3中的main()函数验证这个顶级元素是一个animalList，并且这个nodeToAnimal()函数验证它的参数代表一个精确地带有类型name、species、dateOfBirth、veterinarian和trainer的五个子元素的animal元素。

修改示例14-6是可能的，这样它就能执行这类错误检测。例如，示例14-9中的ContentHandler就验证这个文档的根元素是一个animalList，并且它的子元素是animal类型，animal元素的子元素不包含任何别的元素。它通过维持三个布尔标志：parsingAnimalList_、parsingAnimal_和parsingAnimalChild_，它们记录了在任何时间这个文档正被解析的地方。方法startElement()和endElement()为了一致性简单地更新这些标志并检查它们，它把这个更新当前Animal对象的工作委

www.TopSage.com

托给方法 startAnimalChild()和 endElementChild()，这两个方法的实现与示例14-6中的 startElement()和 endElement()方法相同。

示例 14-9 验证 animals.xml 文档结构的 SAX2 ContentHandler

```
// Implements callbacks which receive character data and
// notifications about the beginnings and ends of elements
class CircusContentHandler : public DefaultHandler {
public:
    CircusContentHandler(vector<Animal>& animalList)
        : animalList_(animalList),    // list to be populated
          parsingAnimalList_(false), // parsing state
          parsingAnimal_(false),     // parsing state
          parsingAnimalChild_(false) // parsing state
        { }

    // Receives notifications from the parser each time
    // beginning of an element is encountered
    void startElement(
            const XMLCh *const uri,       // Namespace uri
            const XMLCh *const localname, // simple tag name
            const XMLCh *const qname,     // qualified tag name
            const Attributes &attrs )     // Collection of attributes
    {
            static XercesString animalList = fromNative("animalList");
            static XercesString animal = fromNative("animal");
            static XercesString xmlns =
                fromNative("http://www.feldman-family-circus.com");

            // Validate the namespace uri
            if (uri != xmlns)
                throw runtime_error(
                          string("wrong namespace uri: ") + toNative(uri)
                     );

            // (i) Update the flags parsingAnimalList_, parsingAnimal_,
            //     and parsingAnimalChild_, which indicate where we are
            //     within the document
            // (ii) verify that the elements are correctly
            //      nested;
            // (iii) Delegate most of the work to the method
            //       startAnimalChild()
            if (!parsingAnimalList_) {
               // We've just encountered the document root
               if (localname == animalList) {
                   parsingAnimalList_ = true; // Update parsing state.
               } else {
                   // Incorrect nesting
                   throw runtime_error(
                             string("expected 'animalList', got ") +
                             toNative(localname )
                         );
               }
            } else if (!parsingAnimal_) {
               // We've just encountered a new animal
               if (localname == animal) {
                   parsingAnimal_ = true;          // Update parsing state.
                   animalList_.push_back(Animal());// Add an Animal to the list.
            } else {
                // Incorrect nesting
                throw runtime_error(
                          string("expected 'animal', got ") +
                          toNative(localname )
                     );
             }
        } else {
```

www.TopSage.com

大家网 TopSage.com

```
            // We're in the middle of parsing an animal element.
            if (parsingAnimalChild_) {
                // Incorrect nesting
                throw runtime_error("bad animal element");
            }
            // Update parsing state.
            parsingAnimalChild_ = true;

            // Let startAnimalChild() do the real work
            startAnimalChild(uri, localname, qname, attrs);
        }
    }


    void endElement(
             const XMLCh *const uri,       // Namespace uri
             const XMLCh *const localname, // simple tag name
             const XMLCh *const qname )    // qualified tag name
    {
       static XercesString animalList = fromNative("animal-list");
       static XercesString animal = fromNative("animal");

       // Update the flags parsingAnimalList, parsingAnimal_,
       // and parsingAnimalChild_; delegate most of the work
       // to endAnimalChild()
       if (localname == animal) {
           parsingAnimal_ = false;
       } else if (localname == animalList) {
           parsingAnimalList_ = false;
       } else {
           endAnimalChild(uri, localname, qname);
           parsingAnimalChild_ = false;
       }
    }

    // Receives notifications when character data is encountered
    void characters(const XMLCh* const chars, const unsigned int length)
    {
        // Append characters to currentText_ for processing by
        // the method endAnimalChild()
        currentText_.append(chars, length);
    }
private:
    // If the current element represents a veterinarian or trainer,
    // use attrs to construct a Contact object for the current
    // Animal; otherwise, clear currentText_ in preparation for the
    // characters() callback
    void startAnimalChild(
             const XMLCh *const uri,       // Namespace uri
             const XMLCh *const localname, // simple tag name
             const XMLCh *const qname,     // qualified tag name
             const Attributes &attrs )     // Collection of attributes
    {
        static XercesString vet = fromNative("veterinarian");
        static XercesString trainer = fromNative("trainer");

        Animal& animal = animalList_.back();
        if (localname == vet) {
            // We've encountered a "veterinarian" element.
            animal.setVeterinarian(contactFromAttributes(attrs));
        } else if (localname == trainer) {
            // We've encountered a "trainer" element.
            animal.setTrainer(contactFromAttributes(attrs));
        } else {
            // We've encountered a "name" , "species", or
            // "dateOfBirth" element. Its content will be supplied
            // by the callback function characters().
```

www.TopSage.com

```
            currentText_.clear();
        }
    }

    // If the current element represents a name, species, or date
    // of birth, use the text stored in currentText_ to set the
    // appropriate property of the current Animal.
    void endAnimalChild(
             const XMLCh *const uri,       // Namespace uri
             const XMLCh *const localname, // simple tag name
             const XMLCh *const qname )    // qualified tag name
    {
        static XercesString name = fromNative("name");
        static XercesString species = fromNative("species");
        static XercesString dob = fromNative("dateOfBirth");

        // currentText_ contains the content of the element which has
        // just ended. Use it to set the current Animal's properties.
        Animal& animal = animalList_.back();
        if (localname == name) {
            animal.setName(toNative(currentText_));
        } else if (localname == species) {
            animal.setSpecies(toNative(currentText_));
        } else if (localname == dob) {
            animal.setDateOfBirth(toNative(currentText_));
        }
    }

    vector<Animal>&  animalList_;          // list to be populated
    bool             parsingAnimalList_;   // parsing state
    bool             parsingAnimal_;       // parsing state
    bool             parsingAnimalChild_; // parsing state
    XercesString     currentText_;         // character data of the
                                           // current text node
};
```

比较示例14-9和示例14-6，你可能注意到了使用回调来验证一个文档的结构是多么复杂。此外，示例14-6仍然没有执行示例14-3中的那些验证：例如，它没有验证一个动物元素的子元素是按正确的顺序出现的，幸运的是，SAX2中还是有更容易的方法来验证一个文档的结构，在14.5节和14.6节你将能看到。

请参考14.1节、14.4节、14.5节和14.6节。

14.4 XML文档操作

问题

你想使用一个C++对象来代表一个XML文档，这样你就可以操作它的元素、属性、文本、DTD、处理指令和注释等。

解决方案

使用W3C DOM Xerces实现。首先，使用类xercesc::DOMImplementationRegistry来获取一个xercesc::DOMImplementation的实例，然后使用这个实例来创建一个xercesc::DOMBuilder解析器的实例。接下来，注册一个xercesc::DOMErrorHandler实例来接收解析错误的通知，并且用你的XML文档的URI或者文件路径作为参数来调用

www.TopSage.com

你的解析器的parseURI()方法。如果这个解析是成功的，parseURI()方法将返回一个指向代表你的XML文档的DOMDocument指针。然后你就可以使用由W3C DOM规范定义的函数来检查和操作你的文档。

当你操作你的XML文档时，你可以通过获取一个来自于DOMImplementation实例的DOMWriter并且使用一个指向DOMDocument的指针作为参数来调用writeNode()方法，从而保存你的XML文档到一个文件中。

示例14-10描述了如何使用DOM来解析示例14-1中的*animals.xml*文档，它定位并删除这个对应于Herby大象的节点，然后保存修改后的文档。

示例14-10 使用DOM来加载、修改和保存XML文档

```
#include <exception>
#include <iostream>// cout
#include <xercesc/dom/DOM.hpp>
#include <xercesc/framework/LocalFileFormatTarget.hpp>
#include <xercesc/sax/SAXException.hpp>
#include <xercesc/util/PlatformUtils.hpp>
#include "animal.hpp"
#include "xerces_strings.hpp"

using namespace std;
using namespace xercesc;

/*
 * Define XercesInitializer as in Example 14-8
 */

// RAII utility that releases a resource when it goes out of scope.
template<typename T>
class DOMPtr {
public:
    DOMPtr(T* t) : t_(t) { }
    ~DOMPtr() { t_->release(); }
    T* operator->() const { return t_; }
private:
    // prohibit copying and assigning
    DOMPtr(const DOMPtr&);
    DOMPtr& operator=(const DOMPtr&);
    T* t_;
};

// Reports errors encountered while parsing using a DOMBuilder.
class CircusErrorHandler : public DOMErrorHandler {
public:
    bool handleError(const DOMError& e)
    {
        std::cout << toNative(e.getMessage()) << "\n";
        return false;
    }
};

// Returns the value of the "name" child of an "animal" element.
const XMLCh* getAnimalName(const DOMElement* animal)
{
    static XercesString name = fromNative("name");

    // Iterate though animal's children
    DOMNodeList* children = animal->getChildNodes();
    for ( size_t i = 0,
```

www.TopSage.com

```
                len = children->getLength();
          i < len;
          ++i )
    {
        DOMNode* child = children->item(i);
        if ( child->getNodeType() == DOMNode::ELEMENT_NODE &&
             static_cast<DOMElement*>(child)->getTagName() == name )
        {
            // We've found the "name" element.
            return child->getTextContent();
        }
    }
    return 0;
}

int main()
{
    try {
        // Initialize Xerces and retrieve a DOMImplementation;
        // specify that you want to  use the Load and Save (LS)
        // feature
        XercesInitializer   init;
        DOMImplementation*  impl =
            DOMImplementationRegistry::getDOMImplementation(
                fromNative("LS").c_str()
            );
        if (impl == 0) {
            cout << "couldn't create DOM implementation\n";
            return EXIT_FAILURE;
        }

        // Construct a DOMBuilder to parse animals.xml.
        DOMPtr<DOMBuilder>  parser =
            static_cast<DOMImplementationLS*>(impl)->
                createDOMBuilder(DOMImplementationLS::MODE_SYNCHRONOUS, 0);

        // Enable namespaces (not needed in this example)
        parser->setFeature(XMLUni::fgDOMNamespaces, true);

       // Register an error handler
       CircusErrorHandler  err;
       parser->setErrorHandler(&err);

       // Parse animals.xml; you can use a URL here
       // instead of a file name
       DOMDocument* doc =
           parser->parseURI("animals.xml");

       // Search for Herby the elephant: first, obtain a pointer
       // to the "animalList" element.
       DOMElement*  animalList = doc->getDocumentElement();
       if (animalList->getTagName() != fromNative("animalList")) {
           cout << "bad document root: "
                << toNative(animalList->getTagName())
                << "\n";
            return EXIT_FAILURE;
    }

    // Next, iterate through the "animal" elements, searching
    // for Herby the elephant.
    DOMNodeList* animals =
        animalList->getElementsByTagName(fromNative("animal").c_str());
    for ( size_t i = 0,
                len = animals->getLength();
          i < len;
          ++i )
    {
        DOMElement* animal =
```

www.TopSage.com

```
            static_cast<DOMElement*>(animals->item(i));
        const XMLCh* name = getAnimalName(animal);
        if (name != 0 && name == fromNative("Herby")) {
            // Found Herby -- remove him from document.
            animalList->removeChild(animal);
            animal->release(); // optional.
            break;
        }
    }

    // Construct a DOMWriter to save animals.xml.
    DOMPtr<DOMWriter> writer =
        static_cast<DOMImplementationLS*>(impl)->createDOMWriter();
    writer->setErrorHandler(&err);

    // Save animals.xml.
    LocalFileFormatTarget file("animals.xml");
    writer->writeNode(&file, *animalList);
} catch (const SAXException& e) {
    cout << "xml error: " << toNative(e.getMessage()) << "\n";
    return EXIT_FAILURE;
} catch (const DOMException& e) {
    cout << "xml error: " << toNative(e.getMessage()) << "\n";
    return EXIT_FAILURE;
} catch (const exception& e) {
    cout << e.what() << "\n";
    return EXIT_FAILURE;
    }
}
```

讨论

如同TinyXml解析器那样，Xerces DOM解析器也产生一个代表XML文档的树结构的C++对象，这个树的节点代表XML文档的组件。Xerces是一个非常复杂的解析器，然而，例如：不像TinyXml，它明白XML的命名空间并且能够处理复杂的DTD。它也能构造代表XML文档的更为详细的结构，包括它的处理指令以及与元素和属性相关的命名空间。更为重要的是，它提供了由W3C DOM规范描述的接口来访问这些信息。

W3C规范还处在发展中，它被分成多个级别；目前有三个级别。类DOMImplementation、DOMDocument、DOMElement和DOMNodeList在示例14-10中使用的都是在DOM第一级中规定的。类DOMBuilder和DOMWrite是在DOM第三级规定的，作为加载和保存的推荐标准的一部分。

注意： Xerces类名并不总是和W3C DOM实现的接口名字相同；因为Xerces在一个单一的命名空间中实现了多个规范，为了避免名字的冲突在一些类名前面加了一些前缀。

示例14-10应该是很容易明白的。就像在示例14-8中一样，我开始时初始化这个Xerces。然后从DOMImplementationRegistry获取DOMImplementation实例，通过传给静态方法DOMImplementationRegistry::getDOMImplementation()一个“LS”字符串来获取加载和存储功能特性。接下来我从DOMIMplementation中获取DOMBuilder。这里我不得不把DOMIMplementation转换成DOMIMplementationLS

www.TopSage.com

类型，因为从W3C DOM的第一级中的接口DOMImplementation是不能访问加载和保存功能特性的。这个createDOMBuilder()的第一个参数表明返回的解析器是以同步方式工作的。Xerces目前还不支持另外一个可能的方式：异步方式。

获取到一个DOMBuilder后，我就打开XML命名空间功能特性，注册一个ErrorHandler并解析这个文档。这个解析器返回一个DOMDocument对象代表这个XML文档；然后我使用DOMDocument的getElementsByTagName()方法，获取一个DOMElement对象，这个DOMElement对象对应于文档中的animalList元素，并且我使用DOMNodeList类型的对象来枚举它的每一个子节点。当我发现一个元素，它的name类型的子元素中包含文本“Herby”时，我就调用根节点元素的removeChild()方法来删除它。

注意： 就像SAX2XMLReader有一个带InputSource实例的方法parse()一样，DOMBuilder也有一个带xercesc::DOMInputSource实例的方法parse()，一个封装了DOMInputSource字符数据源的抽象类有一个具体子类Wrapper4DOMInputSource，这个具体子类可以用来把任意的InputSource转换成xercesc::DOMInputSource。请参考14.3节。

最后我以获取DOMBuilder相同的方式从DOMImplementation中获取一个DOMWriter对象，然后再把这个文档的根节点元素作为参数调用writeNode()方法，把这个修改后的XML文档存储到硬盘中。

注意： 你必须调用release()方法来把由DOMImplementation::createXXX()形式的方法返回的指针释放掉。使用示例14-10中的DOMP tr方法来保证在有异常抛出的情况下也删除这个指针。由DOMDocument::createXXX()形式返回的指针不用显式地删除，尽管当它们不再需要时你也可以删除它们。更详细的信息请参考Xerces类库文档。

14.5 使用DTD验证一个XML文档

问题

如何根据一个DTD来验证一个XML文档？

解决方案

使用Xerces类库中的SAX2（XML的简单API）或者DOM解析器。

为了使用SAX2来验证XML文档，你需要获取一个SAX2XMLReader，如示例14-8中那样。接下来，通过参数xercesc::XMLUni::fgSAX2CoreValidation和true来调用解

www.TopSage.com

析器的setFeature()方法打开DTD合法性验证功能。最后，注册一个ErrorHandler方法来接收DTD违反通知，并且以你的XML文档名作为参数来调用解析器的parse()方法。

为了使用DOM来验证XML文档，首先你需要构造一个XercesDOMParser实例。接下来，以参数xercesc::XercesDOMParser::Val_Always调用解析器的setValidationScheme()方法来打开DTD验证功能。最后，注册一个ErrorHandler类接收DTD违反通知，并且使用你的XML文档名作为参数来调用这个解析器的parse()方法。

注意： 这里我使用类XercesDOMParser，这个解析器是Xerces的一部分，这个部分是在DOM3标准的DOMBuilder接口之前引入的。使用XercesDOMParser更简单，但是你还是可以使用DOMBuilder。请参考14.4节的讨论。

例如，假设你想修改来自示例14-1中的XML文档*animals.xml*使得它包含一个外在的DTD引用，如示例14-11和14-12中说明的那样。使用SAX2 API来验证这个XML文档的合法性的代码在示例14-13中；而使用DOM解析器来验证这个XML文档的代码在示例14-14中。

示例14-11 XML文档animals.xml的DTD animals.dtd文件

```
<!-- DTD for Feldman Family Circus Animals -->

<!ELEMENT animalList (animal+)>
<!ELEMENT animal ( name, species, dateOfBirth,
                   veterinarian, trainer ) >
<!ELEMENT name (#PCDATA)>
<!ELEMENT species (#PCDATA)>
<!ELEMENT dateOfBirth (#PCDATA)>
<!ELEMENT veterinarian EMPTY>
<!ELEMENT trainer EMPTY>
<!ATTLIST veterinarian
    name  CDATA #REQUIRED
    phone CDATA #REQUIRED
>
<!ATTLIST trainer
    name  CDATA #REQUIRED
    phone CDATA #REQUIRED
>
```

示例14-12 修改后的包含DTD的animals.xml文档

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Feldman Family Circus Animals with DTD -->

<!DOCTYPE animalList SYSTEM "animals.dtd">

    <!-- same as Example 14-1 -->
```

www.TopSage.com

```
</animalList>
```

示例 14-13 使用 SAX2 API 来验证 animals.xml 的 DTD 的合法性

```
/*
 * Same includes as Example 14-8, except <vector> is not needed
 */

#include <stdexcept> // runtime_error
#include <xercesc/sax2/DefaultHandler.hpp>

using namespace std;
using namespace xercesc;

/*
 * Define XercesInitializer as in Example 14-8
 * and CircusErrorHandler as in Example 14-7
 */

int main()
{
    try {
        // Initialize Xerces and obtain a SAX2 parser
        XercesInitializer init;
        auto_ptr<SAX2XMLReader>
            parser(XMLReaderFactory::createXMLReader());

        // Enable validation
        parser->setFeature(XMLUni::fgSAX2CoreValidation, true);

        // Register error handler to receive notifications
        // of DTD violations
        CircusErrorHandler error;
        parser->setErrorHandler(&error);
        parser->parse("animals.xml");
    } catch (const SAXException& e) {
        cout << "xml error: " << toNative(e.getMessage()) << "\n";
        return EXIT_FAILURE;
    } catch (const XMLException& e) {
        cout << "xml error: " << toNative(e.getMessage()) << "\n";
        return EXIT_FAILURE;
    } catch (const exception& e) {
        cout << e.what() << "\n";
        return EXIT_FAILURE;
    }
}
```

示例 14-14 使用 XercesDOMParser 来验证这个 animals.xml 文档的 DTD animals.dtd 的合法性

```
#include <exception>
#include <iostream>// cout
#include <stdexcept>    // runtime_error
#include <xercesc/dom/DOM.hpp>
#include <xercesc/parsers/XercesDOMParser.hpp>
#include <xercesc/sax/HandlerBase.hpp>
#include <xercesc/util/PlatformUtils.hpp>
#include "xerces_strings.hpp"  // Example 14-4

using namespace std;
using namespace xercesc;

/*
 * Define XercesInitializer as in Example 14-8
 * and CircusErrorHandler as in Example 14-7
 */
```

www.TopSage.com

```
int main()
{
    try {
        // Initialize Xerces and construct a DOM parser.
        XercesInitializer   init;
        XercesDOMParserparser;

        // Enable DTD validation
        parser.setValidationScheme(XercesDOMParser::Val_Always);

        // Register an error handler to receive notifications
        // of schema violations
        CircusErrorHandler  handler;
        parser.setErrorHandler(&handler);

        // Parse and validate.
        parser.parse("animals.xml");
    } catch (const SAXException& e) {
        cout << "xml error: " << toNative(e.getMessage()) << "\n";
        return EXIT_FAILURE;
    } catch (const XMLException& e) {
        cout << "xml error: " << toNative(e.getMessage()) << "\n";
        return EXIT_FAILURE;
    } catch (const exception& e) {
        cout << e.what() << "\n";
        return EXIT_FAILURE;
    }
}
```

讨论

DTD提供了一种简单的办法来限制XML文档。例如，使用DTD，你可以指定什么样的元素可以出现在一个文档中；一个元素可以有什么样的属性；并且一个特殊的元素是否能包含子元素、文本或者两者都包含等。在元素的子元素的类型、顺序和个数以及属性可以是什么样的值上执行一些限制也是可能的。

DTD的目的是用来标识某个特殊领域的XML文档格式。在示例14-1中，例如，每个动物元素都有子元素name、species、dateofBirth、veterinarian和trainer是很重要的，元素name、species和dateofBirth只包含文本，而veterinarian 和trainer则包含一个name和phone属性。而且，一个动物元素不应该有phone属性，而一个veterinarian元素不应该包含species子元素。

这些都是示例14-11中的DTD实施的限制。例如，接下来的声明表明animal元素按顺序具有name、species、dateOfBirth、veterinarian和trainer元素。

```
<!ELEMENT animal ( name, species, dateOfBirth,
    veterinarian, trainer ) >
```

相同的是，接下来的属性声明表明trainer元素必须有一个name和phone属性；事实上在这个DTD中这个trainer元素没有再声明别的属性，因此也就是说这个元素只具有这两个属性。

```
<!ATTLIST trainer
    name  CDATA #REQUIRED
    phone CDATA #REQUIRED
>
```

www.TopSage.com

如果包含一个DTD的XML文档遵循这个DTD的限制，我们就说这个XML文档是合法的。一个验证XML文档合法性并且验证它的语法错误的XML解析器称为合法性验证解析器。尽管SAX2XMLReader和XercesDOMParser解析器在默认状态下都不是一个合法性验证解析器，但是它们都提供了这种可以打开的这种合法性验证的功能，如示例14-13和14-14所示。相同地，一个DOMBuilder，如在14.4节中说明的那样，通过参数fgXMLUni::fgDOMValidation和true来调用setFeature()方法也可以验证一个XML文档的合法性。

注意： 类SAX2XMLReader、DOMBuilder、DOMWriter和XercesDOMParser都支持一些可选的功能。对于类SAX2XMLReader和 DOMBuilder，你可以使用setFeature()和setProperty()方法来打开或关闭这些功能。第一个方法带一个字符串和一个布尔型作为参数；第二个带一个字符串和一个void*指针作为参数。通过使用getFeature()和getProperty()来查询打开的功能特性。为了方便，Xerces提供了一些常量来代表这些功能特性。类DOMWriter支持setFeature()但不支持setProperty()。而XercesDOMParser类则一个也不支持；它给每个功能特性提供了独立的setter和getter方法。有关这些支持的功能特性的完整描述请参考Xerces文档。

请参考14.6节。

14.6 使用Schema验证一个XML文档

问题

如何根据一个schema来验证一个XML文档的合法性，就像在XML Schema 1.0版本中推荐的那样？

解决方案

使用Xerces类库中的SAX2或者DOM解析器。

使用SAX2 API来使用一个schema验证一个XML文档的合法性与使用DTD来验证一个XML文档的合法性相同，假设一个schema包含在一个目标文档中或者被这个目标文档引用。如果你想使用一个外部的schema来验证一个XML的合法性，你必须调用这个解析器的setProperty()方法来打开这个外部schema的验证特性。这个setProperty()方法的第一个参数就是XMLUni::fgXercesSchemaExternalSchemaLocation或者XMLUni::fgXercesSchemaExternalNoNameSpaceSchemaLocation。依赖这个schema是否有一个目标命名空间。第二个参数应该是这个schema的位置，用一个XMLCh*常量指针来表示。确保这个第二个参数可以转换成void*指针，就像在14.5节中解释的那样。

www.TopSage.com

使用 XercesDOMParser 来验证一个 XML 文档在一个 schema 上的合法性与验证这个 XML 文档在 DTD 上的合法性相同，假设这个 schema 包含在目标文档中或被某个目标文档引用。唯一不同的就是 schema 和命名空间的支持必须显式打开，如示例 14-15 说明的那样。

示例 14-15 使用 XercesDOMParser 来验证一个 XML 文档在一个 schema 上的合法性

```
XercesDOMParser parser;
parser.setValidationScheme(XercesDOMParser::Val_Always);
parser.setDoSchema(true);
parser.setDoNamespaces(true);
```

如果你想验证一个 XML 文档在一个带有目标命名空间的外部 schema 上的合法性，就用你的schema的位置作为参数来调用这个解析器的 setExternalSchemaLocation() 方法。如果你想验证一个 XML 文档在一个不带有命名空间的外部 schema 的合法性，就调用 setExternalNoNamespaceSchema-Location()方法。

相同地，使用 DOMBuilder 来验证一个 XML 文档在一个 schema 上的合法性，必须打开这个验证功能特性，如下所示：

```
DOMBuilder* parser = ...;
parser->setFeature(XMLUni::fgDOMNamespaces, true);
parser->setFeature(XMLUni::fgDOMValidation, true);
parser->setFeature(XMLUni::fgXercesSchema, true);
```

为了使用 DOMBuilder 来验证一个 XML 文档在某个外部 schema 的合法性，给这个 schema 的位置设置 XMLUni::fgXercesSchemaExternalSchemaLocation 或者 XMLUni::fgXercesSchemaExternalNoName-SpaceSchemaLocation 属性。

例如，假设你想使用示例 14-16 中的 schema 来验证示例 14-1 中的 XML 文档 *animals.xml* 的合法性。一个办法就是在这个 *animals.xml* 文档中增加一个 schema 的引用，如示例 14-17 所示。你可以使用 SAX2 API 来验证这个 XML 文档，如示例 14-13 中所示，或者使用 DOM，如示例 14-14 所示，使用示例 14-15 中修改的那样。

示例 14-16 为 SAX2 API 验证的 schema animals.xsd

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Schema for Feldman Family Circus Animals -->

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
<xsd:element name="animalList">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="animal" minOccurs="0" maxOccurs="unbounded">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="name" type="xsd:string"/>
                        <xsd:element name="species" type="xsd:string"/>
                        <xsd:element name="dateOfBirth" type="xsd:date"/>
                        <xsd:element name="veterinarian" type="contact"/>
                        <xsd:element name="trainer" type="contact"/>
                    </xsd:sequence>
```

www.TopSage.com

```
            </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:complexType name="contact">
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="phone" type="phone"/>
</xsd:complexType>

<xsd:simpleType name="phone">
    <xsd:restriction base="xsd:string">
       <xsd:pattern value="\(\d{3}\)\d{3}-\d{4}"/>
    </xsd:restriction>
</xsd:simpleType>

</xsd:schema>
```

示例 14-17 文件 animals.xml，修改后包含一个 schema 的引用

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Feldman Family Circus Animals with Schema -->

<animalList xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:noNamespaceSchemaLocation="animals.xsd">

    <!-- same as Example 14-1 -->

</animalList>
```

另一个办法就是省略这个 schema 引用并且打开外部 schema 验证功能。示例 14-18 中说明 DOM 解析器是如何这样做的。

示例 14-18 使用 DOM 来验证一个 XML 文档在外部 schema 上的合法性

```
/*
 * Same includes as in Example 14-14
 */

using namespace std;
using namespace xercesc;

/*
 * Define XercesInitializer as in Example 14-8
 * and CircusErrorHandler as in Example 14-7
 */

int main()
{
    try {
        // Initialize Xerces and construct a DOM parser.
        XercesInitializer        init;
        XercesDOMParser          parser;

        // Enable validation
        parser.setValidationScheme(XercesDOMParser::Val_Always);
        parser.setDoSchema(true);
        parser.setDoNamespaces(true);
        parser.setExternalNoNamespaceSchemaLocation(
            fromNative("animals.xsd").c_str()
        );

       // Register an error handler to receive notifications
       // of schema violations
```

www.TopSage.com

```
        CircusErrorHandler  handler;
        parser.setErrorHandler(&handler);

        // Parse and validate.
        parser.parse("animals.xml");
    } catch (const SAXException& e) {
        cout << "xml error: " << toNative(e.getMessage()) << "\n";
        return EXIT_FAILURE;
    } catch (const XMLException& e) {
        cout << "xml error: " << toNative(e.getMessage()) << "\n";
        return EXIT_FAILURE;
    } catch (const exception& e) {
        cout << e.what() << "\n";
        return EXIT_FAILURE;
    }
}
```

讨论

就像前几节讨论的DTD那样，schema也是用来限制XML文档的。Schema的目的也是用来标识某个特定领域的定义好的XML文档格式的子集。但schema在三个方面与DTD有所不同。第一，DTD概念和相应的有效性的概念都是用XML规范本身定义的，而schema则是用一个独立的规范描述的，也就是XML schema推荐标准。第二，DTD使用如在示例14-11中描述的特定语法，而schema则使用一个预先定义好的XML文档来表达。第三，schema比DTD有更好的表达性。因为这后面两点，schema比DTD使用得更广。

例如，示例14-11中的DTD要求veterinarian元素只具有两个属性：name和phone，它们的值由字符组成。相反，示例14-16中的schema要求这个phone属性严格地与正则表达式`\(\d{3}\)\d{3}-\d{4}`相匹配，例如，它具有(ddd)xxx-dddd的形式，这里d代表一个任意的数字。相同地，DTD只能需要这个dateOfBirth元素是文本内容，而schema则需要这个文本的格式是yyyy-mm-dd格式，这里yyyy的范围是0001～9999，mm的范围是从01～12，而dd的范围是从01～31。这个能利用这些附加的需求的功能是非常有利的，因为它可以使得原来由程序员完成的工作转成由解析器来完成。

请参考14.5节。

14.7 使用XSLT转换一个XML文档

问题

如何使用一个XSLT样式表stylesheet来转换一个XML文档？

解决方案

使用Xalan类库。首先，构造一个XSLT引擎xalanc::XalanTransformer的实例。接着，构造两个xalanc::XSLTInputSource实例，一个代表这个被转换的XML文档，而另一个则代表你的XSLT样式表，同时还构造一个xalanc::XSLTResultTarget实例来

www.TopSage.com

代表转换后产生的文档。最后，调用XSLT引擎的transform()方法，给它传递XSLTInputSources和XSLTResultTarget作为参数。

例如，假设你想在你的网络浏览器中看到示例14-1中的这些马戏团的动物的话，一个简单的办法就是使用XSLT。示例14-19中说明了一个XSLT样式表，它带一个XML文档（如*animals.xml*）作为输出并且产生一个HTML文档，这个HTML文档包含一个表，表中每个动物占一行，每行包含这个动物的name（名字）、species（种类）、date of birth（出生日期）、veterinarian（兽医）和trainer（培训师）。示例14-20说明了如何使用Xalan类库来把这个样式表应用到*animals.xml*文档上。示例14-20的程序产生的HTML文档说明在示例14-21中，为了可读性在格式上做了一些调整。

示例14-19 XML文档animals.xml的样式表

```
<?xml version="1.0" encoding="utf-8"?>

<!-- Stylesheet for Feldman Family Circus Animals -->

<xsl:stylesheet version="1.1"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="html"/>
    <xsl:template match="/">
        <html>
        <head>
            <title>Feldman Family Circus Animals</title>
        </head>
        <body>
            <h1>Feldman Family Circus Animals</h1>
            <table cellpadding="3" border="1">
                <tr>
                    <th>Name</th>
                    <th>Species</th>
                    <th>Date of Birth</th>
                    <th>Veterinarian</th>
                    <th>Trainer</th>
                </tr>
                <xsl:apply-templates match="animal">
                </xsl:apply-templates>
             </table>
        </body>
        </html>
    </xsl:template>
    <xsl:template match="animal">
        <tr>
            <td><xsl:value-of select="name"/></td>
            <td><xsl:value-of select="species"/></td>
            <td><xsl:value-of select="dateOfBirth"/></td>
            <xsl:apply-templates select="veterinarian"/>
            <xsl:apply-templates select="trainer"/>
        </tr>
    </xsl:template>
    <xsl:template match="veterinarian|trainer">
        <td>
            <table>
                <tr>
                    <th>name:</th>
                    <td><xsl:value-of select="attribute::name"/></td>
                </tr>
                <tr>
                    <th>phone:</th>
                    <td><xsl:value-of select="attribute::phone"/></td>
```

www.TopSage.com

```
            </tr>
          </table>
      </td>
    </xsl:template>
</xsl:stylesheet>
```

示例 14-20 使用 Xalan 类库把 animals.xsl 样式表应用到 animals.xml 文档上

```
#include <exception>
#include <iostream>// cout
#include <xalanc/Include/PlatformDefinitions.hpp>
#include <xalanc/XalanTransformer/XalanTransformer.hpp>
#include <xalanc/XSLT/XSLTInputSource.hpp>
#include <xalanc/XSLT/XSLTResultTarget.hpp>
#include <xercesc/util/PlatformUtils.hpp>
#include "xerces_strings.hpp"  // Example 14-4

using namespace std;
using namespace xercesc;
using namespace xalanc;

// RAII utility that initializes the parser and frees resources
// when it goes out of scope
struct XalanInitializer {
    XalanInitializer()
    {
        XMLPlatformUtils::Initialize();
        XalanTransformer::initialize();
    }
    ~XalanInitializer()
    {
        XalanTransformer::terminate();
        XMLPlatformUtils::Terminate();
    }
};

int main()
{
    try {
        XalanInitializer  init;  // Initialize Xalan.
        XalanTransformer  xslt;                  // XSLT engine.
        XSLTInputSource   xml("animals.xml");    // XML document from
                                                 // Example 14-1.
        XSLTInputSource   xsl("animals.xsl");    // Stylesheet from
                                                 // Example 14-19.
        XSLTResultTarget  html("animals.html"); // xslt's output.

       // Perform transformation.
       if (xslt.transform(xml, xsl, html) != 0) {
           cout << "xml error: " << xslt.getLastError() << "\n";
       }
   } catch (const XMLException& e) {
       cout << "xml error: " << toNative(e.getMessage()) << "\n";
       return EXIT_FAILURE;
   } catch (const exception& e) {
       cout << e.what() << "\n";
       return EXIT_FAILURE;
    }
}
```

示例 14-21 示例 14-20 中的程序产生的 HTML 文档

```
<html>
<head>
    <META http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Feldman Family Circus Animals</title>
</head>
```

www.TopSage.com

大家网 TopSage.com

```
<body>
    <h1>Feldman Family Circus Animals</h1>
    <table cellpadding="3" border="1">
    <tr>
        <th>Name</th>
        <th>Species</th>
        <th>Date of Birth</th>
        <th>Veterinarian</th>
        <th>Trainer</th>
    </tr>
    <tr>
        <td>Herby</td>
        <td>elephant</td>
        <td>1992-04-23</td>
        <td>
            <table>
                <tr><th>name:</th><td>Dr. Hal Brown</td></tr>
                <tr><th>phone:</th><td>(801)595-9627</td></tr>
            </table>
        </td>
        <td>
           <table>
               <tr><th>name:</th><td>Bob Fisk</td></tr>
               <tr><th>phone:</th><td>(801)881-2260</td></tr>
           </table>
        </td>
    </tr>
    <tr>
        <td>Sheldon</td>
        <td>parrot</td>
        <td>1998-09-30</td>
        <td>
            <table>
                <tr><th>name:</th><td>Dr. Kevin Wilson</td></tr>
                <tr><th>phone:</th><td>(801)466-6498</td></tr>
            </table>
         </td>
         <td>
             <table>
                 <tr><th>name:</th><td>Eli Wendel</td></tr>
                 <tr><th>phone:</th><td>(801)929-2506</td></tr>
             </table>
         </td>
    </tr>
    <tr>
        <td>Dippy</td>
        <td>penguin</td>
        <td>2001-06-08</td>
        <td>
            <table>
                <tr><th>name:</th><td>Dr. Barbara Swayne</td></tr>
                <tr><th>phone:</th><td>(801)459-7746</td></tr>
            </table>
        </td>
        <td>
            <table>
                <tr><th>name:</th><td>Ben Waxman</td></tr>
                <tr><th>phone:</th><td>(801)882-3549</td></tr>
            </table>
        </td>
    </tr>
</table>
</body>
</html>
```

讨论

XSL转换（XSLT）是把XML文档转换成其他XML文档的语言。XSLT是规范中的扩展样式语言（XSL）家族中的一个组成部分，它给XML文档可视化提供了一个框架。然而XSLT不只是用于格式化；它可被网络服务器用来在空闲时产生HTML文档，并且它也能被文档产生系统（如DocBook）利用。

作为XML文档表达的XSLT转换称为样式表（stylesheet）。一个样式表在源文档上起作用来产生结果文档。一个样式表由一个模板集合组成，这些模板用来匹配源文档中的节点并且被应用来产生结果文档的某个片断。模板可被递归地应用在源文档中，在结果文档中递增地产生片断直到没有匹配为止。模式匹配受XPath语言的控制，XPath语言是设计来获取信息的，如从XML文档中获取字符串、数字、布尔值和节点等。

示例14-19中的样式表包含三个模板。主要的模板包含一个等于“/”的匹配属性，表明它与源文档的根相匹配，意味着这个节点是这个源文档根节点和任何顶级处理指令和注释的父节点。当这个模板被应用时，它产生一个包含标题为“Feldman Family Circus Animals”的HTML文档的片断，并且这个HTML文档还包含一个表，这个表中的每一行由五个元素组成，而这五个元素的标签分别是Name、Species、Date of Birth、Veterinarian和Trainer。这个模板还包含一个等于animal的匹配属性等应用模板元素。这就产生了样式表的第二个模板 —— 匹配属性animal —— 将会被这个文档根节点的每一个animal子元素上应用一次，给每一个这样的子元素产生HTML文档中表的一行。这个为animal元素产生的行由五个td元素组成。前三个td元素包含animal元素的name、species和dateOfBirth子元素的文本值，使用XSLT的value-of指令来获得这些文本值。后两个td元素包含由应用样式表的第三个模板而创建的表元素 —— 匹配属性veterinarian或trainer —— 也就是匹配animal元素的veterinarian和trainer子元素。

在示例14-20中，尽管我选择给样式表、源文档和结果文档指定当地文件，但是XSLTInputSources和XSLTResultTargets都能从标准的C++类库中的流来构造，让一个XalanTransformer接受任意位置的输入和产生的输出到任意位置。更进一步，无需接受一个XSLTInputSource的实例作为输入，一个XalanTransformer能够操作预编译的样式表和预解析的源文档，这个预编译的样式表由xalanc::XalanCompiledStylesheet的实例来代表，而这个预解析的源文档由xalanc::XalanParsedSource的一个实例来代表。如示例14-22所示。如果你需要把一个单一的样式表应用到多个源文档中，使用XalanCompiledStylesheet比使用XSLTInputSource方便得多。

示例14-22 使用一个预编译好的样式表来执行一个XSLT转换

```
/*
 * Same includes as Example 14-20
 */
```

www.TopSage.com

```
using namespace std;
using namespace xercesc;
using namespace xalanc;

/*
 * Define XalanInitializer as in Example 14-20
 */

int main()
{
    try {
        XalanInitializer    init;   // Initialize Xalan.
        XalanTransformer    xslt;   // XSLT engine.
        XSLTResultTarget    html("animals.html"); // xslt's output.

        // Parse source
        XSLTInputSourcexml("animals.xml");
        XalanParsedSource*  parsedXml = 0;
        if (xslt.parseSource(xml, parsedXml) != 0) {
            cout << "xml error: " << xslt.getLastError() << "\n";
        }

        // Compile stylesheet.
        XSLTInputSource xsl("animals.xsl");
        XalanCompiledStylesheet*  compiledXsl = 0;
        if (xslt.compileStylesheet(xsl, compiledXsl) != 0) {
            cout << "xml error: " << xslt.getLastError() << "\n";
        }

       // Perform transformation.
       if (xslt.transform(xml, xsl, html)) {
           cout << "xml error: " << xslt.getLastError() << "\n";
       }
   } catch (const XMLException& e) {
       cout << "xml error: " << toNative(e.getMessage()) << "\n";
       return EXIT_FAILURE;
   } catch (const exception& e) {
       cout << e.what() << "\n";
       return EXIT_FAILURE;
   }
}
```

请参考14.8节。

14.8 评估一个XPath表达式

问题

如何通过评估一个XPath表达式来从一个解析过的XML文档中获取信息？

解决方案

使用Xalan类库。首先，解析这个XML文档以获得一个指向一个xalanc::XalanDocument实例的指针。通过使用XalanSourceTreeInit、XalanSourceTreeDOMSupport和XalanSourceTreeParserLiaison实例就能实现，这三个类都定义在xalanc命名空间中，如下所示：

```
#include <xercesc/framework/LocalFileInputSource.hpp>
#include <xalanc/XalanSourceTree/XalanSourceTreeDOMSupport.hpp>
```

www.TopSage.com

```
#include <xalanc/XalanSourceTree/XalanSourceTreeInit.hpp>
#include <xalanc/XalanSourceTree/XalanSourceTreeParserLiaison.hpp>
...
int main()
{
    ...
    // Initialize the XalanSourceTree subsystem
    XalanSourceTreeInit init;
    XalanSourceTreeDOMSupportsupport;

    // Interface to the parser
    XalanSourceTreeParserLiaison  liaison(support);

    // Hook DOMSupport to ParserLiaison
    support.setParserLiaison(&liaison);
    LocalFileInputSource          src(document-location);
    XalanDocument*                doc = liason.ParseXMLStream(doc);
    ...
}
```

同时，你也可以使用 Xerces DOM 解析器来获得一个指向 DOMDocument 实例的指针，如示例 14-14 那样，然后使用 XercesDOMSupport、XercesParserLiaison 和 XercesDOMWrapperParsedSource 的实例（这些类也定义在 xalanc 命名空间中）获取一个对应于这个 DOMDocument 的 XalanDocument 的指针。

```
#include <xercesc/dom/DOM.hpp>
#include <xalanc/XalanTransformer/XercesDOMWrapperParsedSource.hpp>
#include <xalanc/XercesParserLiaison/XercesParserLiaison.hpp>
#include <xalanc/XercesParserLiaison/XercesDOMSupport.hpp>
...
int main() {
    ...
    DOMDocument*                  doc = ... ;
    XercesDOMSupport              support;
    XercesParserLiaison           liaison(support);
    XercesDOMWrapperParsedSource  src(doc, liaison, support);
    XalanDocument*                xalanDoc = src.getDocument();
    ...
}
```

接下来，当你评估这个 XPath 表达式时，获取一个指向作为内容节点的节点的指针。你可以使用 XalanDocument 的 DOM 接口来获取到。构造一个 XPathEvaluator 来评估这个 XPath 表达式，并构造一个 XalanDocumentPrefixResolver 来解析在这个 XML 文档中的命名空间前缀。最后，调用 XPathEvaluator 的 evaluate() 方法，把这个 DOMSupport、内容节点、这个 XPath 表达式和这个 PrefixResolver 作为参数传入这个方法。评估这个表达式的结果将作为类型 XObjectPtr 的一个对象返回；你能在这个对象上执行的操作依赖于这个 XPath 数据的类型，而这个 XPath 数据类型你可以使用 getType() 方法来获得。

例如，假设你想获取示例 14-1 中的 *animals.xml* 文档的动物的名字的话。你就可以通过解析这个文档并且使用这个文档的根节点作为内容节点，来评估这个 XPath 表达式 animalList/animal/name/child::text() 以获取动物名字信息。如下面的示例 14-23 所示。

www.TopSage.com

示例 14-23 使用 Xalan 来评估一个 XPath 表达式

```
#include <cstddef>        // size_t
#include <exception>
#include <iostream>       // cout
#include <xercesc/dom/DOM.hpp>
#include <xercesc/parsers/XercesDOMParser.hpp>
#include <xercesc/sax2/DefaultHandler.hpp>
#include <xercesc/util/PlatformUtils.hpp>
#include <xalanc/DOMSupport/XalanDocumentPrefixResolver.hpp>
#include <xalanc/XalanTransformer/XercesDOMWrapperParsedSource.hpp>
#include <xalanc/XercesParserLiaison/XercesParserLiaison.hpp>
#include <xalanc/XercesParserLiaison/XercesDOMSupport.hpp>
#include <xalanc/XPath/XObject.hpp>
#include <xalanc/XPath/XPathEvaluator.hpp>
#include "animal.hpp"
#include "xerces_strings.hpp"

using namespace std;
using namespace xercesc;
using namespace xalanc;

// RAII utility that initializes the parser and the XPath engine
// and frees resources when it goes out of scope
class XPathInitializer {
public:
    XPathInitializer()
    {
        XMLPlatformUtils::Initialize();
        XPathEvaluator::initialize();
    }
        ~XPathInitializer()
    {
        XPathEvaluator::terminate();
        XMLPlatformUtils::Terminate();
    }
private:
    // Prohibit copying and assignment
    XPathInitializer(const XPathInitializer&);
    XPathInitializer& operator=(const XPathInitializer&);
};

// Receives Error notifications
class CircusErrorHandler : public DefaultHandler {
public:
    void error(const SAXParseException& e)
    {
        throw runtime_error(toNative(e.getMessage()));
    }
    void fatalError(const SAXParseException& e) { error(e); }
};

int main()
{
    try {
        // Initialize Xerces and XPath and construct a DOM parser.
        XPathInitializer    init;
        XercesDOMParser     parser;

        // Register error handler
        CircusErrorHandler error;
        parser.setErrorHandler(&error);

        // Parse animals.xml.
        parser.parse(fromNative("animals.xml").c_str());
        DOMDocument* doc = parser.getDocument();
        DOMElement*  animalList = doc->getDocumentElement();
```

www.TopSage.com

大家网 TopSage.com

```
        // Create a XalanDocument based on doc.
        XercesDOMSupport                support;
        XercesParserLiaison             liaison(support);
        XercesDOMWrapperParsedSource    src(doc, liaison, support);
        XalanDocument*                  xalanDoc = src.getDocument();

        // Evaluate an XPath expression to obtain a list
        // of text nodes containing animals' names
        XPathEvaluator                  evaluator;
        XalanDocumentPrefixResolver     resolver(xalanDoc);
        XercesString                    xpath =
           fromNative("animalList/animal/name/child::text()");
        XObjectPtr                      result =
            evaluator.evaluate(
                support,        // DOMSupport
                xalanDoc,       // context node
                xpath.c_str(),  // XPath expr
                resolver );     // Namespace resolver
        const NodeRefListBase&          nodeset = result->nodeset();


        // Iterate through the node list, printing the animals' names
        for ( size_t i = 0,
                     len = nodeset.getLength();
              i < len;
              ++i )
        {
              const XMLCh* name =
                nodeset.item(i)->getNodeValue().c_str();
              std::cout << toNative(name) << "\n";
        }
    } catch (const DOMException& e) {
        cout << "xml error: " << toNative(e.getMessage()) << "\n";
        return EXIT_FAILURE;
    } catch (const exception& e) {
        cout << e.what() << "\n";
        return EXIT_FAILURE;
    }
}
```

讨论

XPath是一种模式匹配语言，其设计是用来从XML文档中获取信息。XPath的主要结构——路径表达式——提供了一个层次语法，基于它们的名字、属性、文本内容、继承关系以及其他的一些属性来引用元素、属性和文本节点。除了在节点或节点集上执行一些操作，XPath语言还能处理字符串、数字和布尔值。目前Xalan类库还不支持XPath 2.0版本，它基于XML Schema推荐标准提供了更为丰富的数据模型（请参考14.5节）。

XPath表达式在XML文档的内容节点中被评估，这个内容节点是用来阐述相关的结构的，如父亲、儿子和后代等。示例14-23中，我把这个XML文档的根作为这个内容节点，这个根是这个XML文档的根元素和任何顶级处理指令和注释的父亲。当以这个根节点作为内容节点评估时，这个路径表达式`animalList/animal/name/child::text()`将匹配`name`元素的所有文本子节点，它的父元素是`animal`元素，而它的祖父元素是`animalList`元素。

`XPathEvaluator`的`evaluate()`方法返回一个`XObjectPtr`，它代表对一个XPath表达式评估的结果。`XObjectPtr`的数据类型通过间接引用它获得一个`XObject`并调用

www.TopSage.com

getType()方法可以获得；然后它下面的数据通过调用num()、boolean()、str()或nodeset()方法就可以获得。由于示例14-23中的XPath表达式代表一个节点集，因此我使用nodeset()方法来获得一个NodeRefListBase的引用，通过它的getLength()方法和item()方法你就可以访问它的每一个节点。方法item()返回一个指向XalanNode的指针，它的getNodeValue()方法返回一个字符串，这个字符串的接口与std::basic_string的接口相同。

由于XPath提供了一个很容易的办法来定位一个XML文档中的节点，很自然地，你可能想使用Xalan的XPath表达式来获取来自xercesc::DOMDocument中的xercesc::DOMNode一个实例。事实上，这是可能的，但是稍有点笨拙；更为重要的是，默认的情况下，通过这种方式获取的xercesc::DOMNode只是这个XML文档树的只读的一部分，这就限制了XPath作为工具来操作DOM。当然还是有办法绕开这个限制的，但是比较复杂并有一些潜在的危险。

幸运的是，这个Pathan类库提供了与Xerces相兼容的XPath实现，这样你就很容易地操作Xerces DOM。示例14-24说明了如何使用Pathan来定位和删除这个来自示例14-1的XML文档中对应的Herby大象节点，它是通过评估XPath表达式animalList/animal[child::name='Herby']来实现的。与示例14-10相比，你就会发现XPath语言是多么清晰和强大。

示例14-24 使用Pathan来定位一个节点并把它删除

```
#include <exception>
#include <iostream>// cout
#include <xercesc/dom/DOM.hpp>
#include <xercesc/framework/LocalFileFormatTarget.hpp>
#include <xercesc/util/PlatformUtils.hpp>
#include <pathan/XPathNamespace.hpp>
#include <pathan/XPathResult.hpp>
#include <pathan/XPathEvaluator.hpp>
#include <pathan/XPathExpression.hpp>
#include "xerces_strings.hpp"  // Example 14-4

using namespace std;
using namespace xercesc;

/*
 * Define XercesInitializer as in Example 14-8, and
 * CircusErrorHandler and DOMPtr as in Example 14-10
 */

int main()
{
    try {
        // Initialize Xerces and retrieve a DOMImplementation.
        XercesInitializer   init;
        DOMImplementation*  impl =
            DOMImplementationRegistry::getDOMImplementation(
                fromNative("LS").c_str()
            );
       if (impl == 0) {
           cout << "couldn't create DOM implementation\n";
           return EXIT_FAILURE;
       }
```

www.TopSage.com

```
      // Construct a DOMBuilder to parse animals.xml.
      DOMPtr<DOMBuilder>  parser =
          static_cast<DOMImplementationLS*>(impl)->
              createDOMBuilder(
                  DOMImplementationLS::MODE_SYNCHRONOUS, 0
              );
      CircusErrorHandler  err;
      parser->setErrorHandler(&err);

      // Parse animals.xml.
      DOMDocument*  doc =
          parser->parseURI("animals.xml");
      DOMElement*   animalList = doc->getDocumentElement();

      // Create XPath expression.
      auto_ptr<XPathEvaluator>
          evaluator(XPathEvaluator::createEvaluator());
      auto_ptr<XPathNSResolver>
          resolver(evaluator->createNSResolver(animalList));
      auto_ptr<XPathExpression>
          xpath(
              evaluator->createExpression(
                  fromNative(
                      "animalList/animal[child::name='Herby']"
                  ).c_str(),
                  resolver.get()
              )
          );
auto_ptr<XPathEvaluator>   evaluator(XPathEvaluator::createEvaluator());
auto_ptr<XPathNSResolver>  resolver(evaluator->createNSResolver(animalList));
auto_ptr<XPathExpression>  xpath(
      evaluator->createExpression(
            fromNative("animalList/animal[child::name='Herby']").c_str(),
            resolver.get()
     ));

       // Evaluate the expression.
       XPathResult* result =
           xpath->evaluate(
               doc,
               XPathResult::ORDERED_NODE_ITERATOR_TYPE,
               0
           );

      DOMNode* herby;
      if (herby = result->iterateNext()) {
          animalList->removeChild(herby);
          herby->release(); // optional.
      }

      // Construct a DOMWriter to save animals.xml.
      DOMPtr<DOMWriter> writer =
         static_cast<DOMImplementationLS*>(impl)->createDOMWriter();
       writer->setErrorHandler(&err);

       // Save animals.xml.
       LocalFileFormatTarget file("circus.xml");
       writer->writeNode(&file, *animalList);
   } catch (const DOMException& e) {
       cout << toNative(e.getMessage()) << "\n";
       return EXIT_FAILURE;
   } catch (const XPathException &e) {
       cout << e.getString() << "\n";
       return EXIT_FAILURE;
   } catch (const exception& e) {
       cout << e.what() << "\n";
       return EXIT_FAILURE;
```

www.TopSage.com

```
    }
}
```

示例 14-24 中我使用的是 Pathan 1，它实现了 XPath 1.0 的推荐标准，同样这个 1.0 的标准在 Xalan 库中也支持。目前，可以获得 Pathan 2 的 beta 版本了，它提供了 XPath 2.0 推荐标准的初步实现。Pathan 2 代表了 XPath 标准的一个更可信赖的实现；一旦这个 Pathan 2 的非 beta 版本可以获得的话，我推荐你还是使用 Pathan 2。

请参考 14.7 节。

14.9 使用 XML 保存和还原对象集

问题

如何把一个 C++ 集合中的元素存储到一个 XML 文档中，并且以后还能从这个 XML 文档再把它们读入内存？

解决方案

使用 Boost 的序列化类库。这个库让你可以使用一个称为 *archives* 的类来存储对象。为了利用这个库，你必须首先使得你的每一个类都是可以序列化的，这就意味着你可以把你的类的所有实例都写入到一个存档中去，或者称为序列化，并且以后还可以从这个存档中读入内存，或者叫做反序列化。然后，在运行时，你就可以使用 << 操作符来把你的对象存到存档中并且使用 >> 操作符把这些对象从存档中读入内存。

为了使得一个类是可序列化的，你需要给你的类增加一个成员函数模板 `serialize`，其签名如下所示：

```
template<typename Archive>
void serialize(Archive& ar, const unsigned int version);
```

这个 `serialize` 函数模板的实现应该把你的类的每个数据成员作为名－值对写到这个指定的 archive 中去，使用 & 操作符。例如，如果你想序列化和反序列化示例 14-2 中的 `Contact` 类的实例，就增加一个成员函数 `serialize`，如示例 14-25 所示。

示例 14-25 给示例 14-2 中的 Contact 类增加序列化支持

```
#include <boost/serialization/nvp.hpp> // "name-value pair"

class Contact {
...
private:
    friend class boost::serialization::access;
    template<typename Archive>
    void serialize(Archive& ar, const unsigned int version)
    {
        // Write (or read) each data-member as a name-value pair
        using boost::serialization::make_nvp;
        ar & make_nvp("name", name_);
        ar & make_nvp("phone", phone_);
```

www.TopSage.com

```
    }
    ...
};
```

相同地，你也可以给示例 14-2 中的 Animal 类增加序列化支持，如示例 14-26 所示。

示例 14-26 给示例 14-2 中的 Animal 增加序列化支持

```
...
// Include serialization support for boost::gregorian::date
#include <boost/date_time/gregorian/greg_serialize.hpp>
...
class Contact {
...
private:
    friend class boost::serialization::access;
    template<typename Archive>
    void serialize(Archive& ar, const unsigned int version)
    {
        // Write (or read) each data-member as a name-value pair
        using boost::serialization::make_nvp;
        ar & make_nvp("name", name_);
        ar & make_nvp("species", species_);
        ar & make_nvp("dateOfBirth", dob_);
        ar & make_nvp("veterinarian", vet_);
        ar & make_nvp("trainer", trainer_);
    }
...
};
```

现在，通过创建一个类型为 boost::archive::xml_oarchive 的 XML 存档你就可以序列化一个 Animal 类，并且通过使用 << 操作符来把一个 Animal 类的实例写入这个存档中。这个 xml_oarchive 构造函数带一个 std::ostream 作为参数；常常这是一个写入文件的输出流，但总的来说，它可以是写入到任何类型的数据源的流。当一个 Animal 实例被序列化完后，通过构造一个类型为 boost::archive::xml_iarchive 的 XML 存档就可以把它读入内存，这个 XML 存档连接与原来的那个存档相同的数据源，并且你需要调用 >> 操作符。

示例 14-27 说明如何使用 Boost 的序列化库来把这个 Animal 的 std::vector 写入到文件 *animals.xml* 中并且再把它们载入内存。示例 14-27 中运行完这个程序的 *animals.xml* 的内容说明在示例 14-28 中。

示例 14-27 序列化一个 Animal 类型的 std::vector

```
#include <fstream>
#include <boost/archive/xml_oarchive.hpp> // Archive for writing XML
#include <boost/archive/xml_iarchive.hpp> // Archive for reading XML
#include <boost/serialization/vector.hpp> // machinery for serializing
#include "animal.hpp"                     // std::vector

int main()
{
    using namespace std;
    using namespace boost::archive;       // namespace for archives
    using namespace boost::serialization; // namespace for make_nvp

    try {
        // Populate list of animals
```

www.TopSage.com

大象网 TopSage.com

```
        vector<Animal> animalList;
        animalList.push_back(
            Animal( "Herby", "elephant", "1992-04-23",
                    Contact("Dr. Hal Brown", "(801)595-9627"),
                    Contact("Bob Fisk", "(801)881-2260") ));
        animalList.push_back(
            Animal( "Sheldon", "parrot", "1998-09-30",
                    Contact("Dr. Kevin Wilson", "(801)466-6498"),
                    Contact("Eli Wendel", "(801)929-2506") ));
        animalList.push_back(
            Animal( "Dippy", "penguin", "2001-06-08",
                    Contact("Dr. Barbara Swayne", "(801)459-7746"),
                    Contact("Ben Waxman", "(801)882-3549") ));

        // Construct XML output archive and serialize list
        ofstream  fout("animals.xml");
        xml_oarchive   oa(fout);
        oa << make_nvp("animalList", animalList);
        fout.close();

        // Construct XML intput archive and deserialize list
        ifstream        fin("animals.xml");
        xml_iarchive    ia(fin);
        vector<Animal>  animalListCopy;
        ia >> make_nvp("animalList", animalListCopy);
        fin.close();

        if (animalListCopy != animalList) {
            cout << "XML serialization failed\n";
            return EXIT_FAILURE;
        }

    } catch (const exception& e) {
        cout << e.what() << "\n";
        return EXIT_FAILURE;
    }
}
```

示例 14-28 示例 14-27 中的程序运行完后的 animals.xml 文件

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="3">
<animalList class_id="0" tracking_level="0" version="0">
    <count>3</count>
    <item class_id="1" tracking_level="0" version="0">
        <name>Herby</name>
        <species>elephant</species>
        <dateOfBirth class_id="2" tracking_level="0" version="0">
            <date>19920423</date>
        </dateOfBirth>
        <veterinarian class_id="3" tracking_level="0" version="0">
            <name>Dr. Hal Brown</name>
            <phone>(801)595-9627</phone>
        </veterinarian>
        <trainer>
            <name>Bob Fisk</name>
            <phone>(801)881-2260</phone>
        </trainer>
    </item>
    <item>
        <name>Sheldon</name>
        <species>parrot</species>
        <dateOfBirth>
            <date>19980930</date>
        </dateOfBirth>
        <veterinarian>
```

www.TopSage.com

```
            <name>Dr. Kevin Wilson</name>
            <phone>(801)466-6498</phone>
        </veterinarian>
        <trainer>
            <name>Eli Wendel</name>
            <phone>(801)929-2506</phone>
        </trainer>
    </item>
    <item>
        <name>Dippy</name>
        <species>penguin</species>
        <dateOfBirth>
             <date>20010608</date>
        </dateOfBirth>
        <veterinarian>
            <name>Dr. Barbara Swayne</name>
            <phone>(801)459-7746</phone>
        </veterinarian>
        <trainer>
            <name>Ben Waxman</name>
           <phone>(801)882-3549</phone>
        </trainer>
    </item>
</animalList>
```

讨论

Boost序列化库提供了最广泛和灵活的方法来保存和恢复C++对象。它是非常复杂的框架；例如，你可以序列化复杂的数据结构，这个数据结构包含指向多态对象的循环的引用和指针。更为重要的是，这个库在XML文档序列之外也是有用的：除了XML的存档，它也给文本和二进制的存档提供了多个类型。XML和文本的存档是可以移植的，意味着这些数据可以在一个系统中序列化并且在另一个系统中反序列化；而二进制的存档则是不可移植的，但它提供了压缩功能。

由Boost序列化库产生的XML文档不遵循任何前面存在的规范，并且它的格式在将来的Boost版本中也可能会改变。因此，你不可以使用这些文档来连接别的C++序列化框架。虽然如此，XML序列化还是有用的，因为序列化的输出对于人来说是很容易读的，并且这些输出也可以被XML处理工具处理。

示例14-25和14-26说明了插入的序列化：类`Animal`和`Contact`都被修改来支持序列化。Boost序列化库也支持非插入的序列化，它使得不用修改类的定义就能使得类能够序列化，通过一些公共的接口来访问对象的状态。你已经在示例14-27中看到了关于非插入序列化的例子：这个模板`std::vector`是可以被序列化的，尽管它的定义不能被终端用户修改。事实上，所有标准容器都是可以被序列化的；为了使得定义在标准头文件*xxx*中的容器可以被序列化，你需要简单地包含头文件*boost/serialization/xxx.hpp*。更多的关于非插入序列化的信息，请参考Boost序列化文档。

示例14-25和示例14-26说明操作符`&`的双重作用：当一个对象正被序列化时它就像`<<`操作符，而当一个对象被反序列化时它就像`>>`操作符。这是方便的，因为这让你把序列化和反序列化实现为一个简单的函数。然而在某些情况下，使用一个简单的函数来实现

www.TopSage.com

序列化和反序列化则不是很合适；这些情况下，Boost.Serialization序列化库提供了一种机制把这个serialize()方法分解成独立的load()和save()方法。如果你需要利用这个功能，请参考Boost.Serialization相应的文档。

示例14-25、14-26和14-27中，我使用了函数boost::serialization::make_nvp构造名/值对。Boost.Serialization也提供了一个宏BOOST_SERIALIZATION_NVP让你仅提供一个变量的名字来序列化这个变量。这个名/值对的第一个组件将会由预处理器来自动构造，使用操作符#来把这个宏参数转换成字符串常量。

```
// Same as ar & make_nvp("name_", name_);
ar & BOOST_SERIALIZATION_NVP(name_);
```

在这个例子中，我用make_nvp代替BOOST_SERIALIZATION_NVP宏将能更多地控制标签的名字，使得这个XML存档的内容更容易读。

当给从其他可以序列化的类派生的类增加序列化支持时，Boost.Serialization序列化文档建议这个serialize()声明为私有的，这样可以减少用户的错误。为了让Boost.Serialization序列化库能够调用你的类的serialize()方法，你必须把这个类boost::serialization::access声明为你的类的友元。

最后，示例14-25和14-26中serialize()的第二个参数是Boost.Serialization序列化支持的类的版本的一部分。第一次将某个类的一个对象存储到一个archive中时，这个类的版本也被存储；当这个类的一个实例被反序列化时，Boost.Serialization序列化把这个存储的版本作为第二个参数传给serialize方法。这个信息能用于自定义的反序列化；例如，仅当这个archive中记录的类的版本不比这个类中声明的变量的第一个版本低时反序列化可以加载一个成员变量的值。默认的情况下，一个类的版本是0。为了详细说明一个类的版本，把这个类的名字和类的版本作为参数调用宏BOOST_CLASS_VERSION，这个宏定义在头文件*boost/serialization/version.hpp*中。

www.TopSage.com

第15章

杂项

15.0 概述

本章将要讨论C++中不属于前面那些章节的主题：函数和成员指针、常量型成员函数和成员变量以及独立（例如，非成员）操作符和一些别的主题。

15.1 为回调使用函数指针

问题

你打算调用某个函数fun1，同时在运行时你又想调用另外一个函数fun2。由于某种原因，你不能把函数fun2硬编码到fun1函数中。编译时，函数fun2不可能被确定，或者函数fun1属于某个第三方的API，你不可能改变和编译它。也就是说，在这些情况下，如何使用回调函数？

解决方案

在以上的这些函数例子中，你可以把函数fun1声明为带一个函数指针作为参数的函数，并且把运行时函数fun2的地址传给它。再使用typedef重新定义一下这个函数类型以使得它更容易读和调试。示例15-1描述了使用函数指针来实现一个回调函数。

示例15-1 回调函数

```
#include <iostream>

// An example of a callback function
bool updateProgress(int pct) {

   std::cout << pct << "% complete...\n";
   return(true);
}

// A typedef to make for easier reading
typedef bool (*FuncPtrBoolInt)(int);
```


www.TopSage.com

```
// A function that runs for a while
void longOperation(FuncPtrBoolInt f) {

   for (long l = 0; l < 100000000; l++)
      if (l % 10000000 == 0)
         f(l / 1000000);
}

int main() {

   longOperation(updateProgress); // ok
}
```

讨论

在示例15-1中，当updateProgress和longOperation函数相互之间都不知道的情况下，使用函数指针是非常合适的。例如，一个通过把进度显示在控制窗口中的用户界面对话框中的更新进度的函数，它在别的不关心进度内容的地方被调用。同样，longOperation函数可能是那些数据载入函数的一部分，这些载入函数可能是在图形用户界面、控制窗口或者其他的后台进程中被调用。

你需要做的第一件事情就是决定函数的原型，然后使用typedef操作符重新定义它。当你碰到函数指针时，typedef是一个很好的重新定义的操作，因为函数指针一般很不好懂。考虑一下你将如何来声明一个函数指针变量f，它是一个带一个简单整型参数并返回一个布尔型值的函数的地址。如下所示：

```
bool (*f)(int);  // f is the variable name
```

看到这个定义，也许你会争辩，只不过如此而已，这不难。但如果你需要这样一个函数指针的矢量，该怎办？

```
vector<bool (*)(int)> vf;
```

它们是个数组吗？

```
bool (*af[10])(int);
```

函数指针的声明不像平常的C ++的变量声明那样在一个有效类型后跟一个变量名就可以。这就是为什么它们不好读懂的原因。

因此，示例15-1中，使用了typedef重新定义如下：

```
typedef bool (*FuncPtrBoolInt)(int);
```

一旦这样定义好了，我就能很容易地和下面一样定义一个返回一个布尔型值并接受一个简单整型值的函数类型，就像使用其他类型的参数那样：

```
void longOperation(FuncPtrBoolInt f) {
   // ...
```

现在，所有的longOperation函数就像调用别的任何其他函数那样调用f。

```
f (l/1000000);
```

www.TopSage.com

大家网 TopSage.com

这样，f参数就可以是任何只要接受一个整型数作为参数并返回一个布尔型值的函数。如果longOperation的调用者不关心进度情况，它就可以用一个没有操作的函数传给它：

```
bool whoCares(int i) {return(true);}
//...
longOperation(whoCares);
```

更为重要的是，传给 longOperation 函数可以在运行时再确定。

15.2 使用指向类成员的指针

问题

如何使用地址来引用数据成员或成员函数？

解决方案

使用类名和范围操作符(::)加上星号来正确地引用类成员。示例 15-2 给出了一个简单的例子。

示例 15-2 获取一个指向成员的指针

```
#include <iostream>
#include <string>

class MyClass {

public:
   MyClass() : ival_(0), sval_("foo") {}
  ~MyClass() {}

   void incr() {++ival_;}
   void decr() {ival_--;}

private:
   std::string sval_;
   int ival_;
};

int main() {

   MyClass obj;

   int         MyClass::* mpi = &MyClass::ival_;  // Data member
   std::string MyClass::* mps = &MyClass::sval_;  // pointers

   void (MyClass::*mpf)(); // A pointer to a member function that
                           // takes no params and returns void
   void (*pf)();           // A normal function pointer

   int* pi = &obj.ival_;   // int pointer referring to int member--no
                           // problem.

   mpf = &MyClass::incr;   // A pointer to a member function. You can't
                           // write this value to a stream. Look at it
                           // in your debugger to see what its
                           // representation looks like.

   pf = &MyClass::incr;    // Error: &MyClass::incr is not an instance
                           // of a function
```

www.TopSage.com

```
    std::cout << "mpi = " << mpi << '\n';
    std::cout << "mps = " << mps << '\n';
    std::cout << "pi =  " << pi << '\n';
    std::cout << "*pi = " << *pi << '\n';

    obj.*mpi = 5;
    obj.*mps = "bar";

    (obj.*mpf)(); // now obj.ival_ is 6

    std::cout << "obj.ival_ = " << obj.ival_ << '\n';
    std::cout << "obj.sval_ = " << obj.sval_ << '\n';
}
```

讨论

指向成员的指针和普通的指针有很大的不同。首先，它们具有很有趣的语法。请看示例15-2中的如下这行代码：

```
int MyClass::* mpi = &MyClass::ival_;
```

这一行代码在声明的同时赋值了一个整型指针，它指向类MyClass中的一个成员。在两个方面使得它和普通的指向整型数的指针int*不同。首先，在数据类型和星号之间，你不得不包括类名和范围操作符。第二，当你给这个指针赋值时，你不是赋给它一个内存中的地址。&MyClass::ival_事实上不是一个内存中的具体地址，也不可能是；它引用类名而不是对象名，这是为什么呢？把它想象成是一个从这个类对象的起始地址开始到这个数据成员之间的一个偏移量就可以。

Mpi变量不得不与它对应类的一个对象一起使用。示例15-2中的如下这行代码中，使用一个整型值5赋给mpi所指的对象：

```
obj.*mpi = 5;
```

obj是类MyClass的一个实例。通过使用点号（如果obj是一个指针，则使用->）可以引用它的成员并且和mpi做个比较，你可以得到一个索引指向obj.ival_。

指向成员函数的指针基本相同。示例15-2中声明了一个指向MyClass类成员函数的指针，这个成员函数没带参数并返回void类型的值。

```
void (MyClass::*mpf)();
```

使用取地址符给它赋一个值：

```
mpf = &MyClass::incr;
```

为了调用它，你可以如下面这行代码那样给它加上括号：

```
(obj.*mpf)();
```

数据成员指针和成员函数指针还有点区别。你如果想用一个普通的非成员指针指向一个数据成员，你可以如下这样做：

```
int* pi = &obj.ival_;
```

www.TopSage.com

当然，你应使用一个对象名而不是类名，因为你是要得到内存中的一个具体对象的某个数据成员的（典型地，你不应该外露你的类的数据成员的地址，免得被客户的代码无意中修改了它的值）。

相反，在成员函数上，你不应该这样做，因为这没有任何意义。考虑一个函数指针，假设它与MyClass::incr成员函数具有同样的原型(例如,不带有参数并返回void类型值)：

```
void (*pf)();
```

现在，试着把成员函数的地址赋给它：

```
pf = &MyClass::incr; // Nope
pf = &obj.incr;      // No dice
```

以上两个都不能通过编译。一个成员函数需要对象的上下文才有意义，因为它需要引用成员变量。不在任何对象中调用一个成员函数需要这个成员函数不使用对象的任何成员变量，这大概就是一个成员函数是成员函数而不是一个单独的函数的原因吧。

请参考15.1节。

15.3 确保函数不会修改参数

问题

如何编写一个函数，以保证它的参数在被调用时不被修改？

解决方案

用关键字const来声明这个参数就可以防止参数被修改。示例15-3给出了一个简单的例子。

示例15-3 确保不修改参数

```
#include <iostream>
#include <string>

void concat(const std::string& s1, // These are declared const, so they
            const std::string& s2, // cannot be changed
            std::string& out) {
   out = s1 + s2;
}

int main() {

   std::string s1 = "Cabo ";
   std::string s2 = "Wabo";
   std::string s3;

   concat(s1, s2, s3);

   std::cout << "s1 = " << s1 << '\n';
   std::cout << "s2 = " << s2 << '\n';
   std::cout << "s3 = " << s3 << '\n';
}
```

www.TopSage.com

讨论

示例15-3中给出了const的一种直观用法。当你不想在调用某个函数时修改它的参数时，有很多好的原因来解释应把你的函数参数声明为const。首先，你给你的代码的读者传递了你的意图。通过把参数声明为const，你告诉了读者这个参数只是为输入才用的。这就让你的代码的用户在使用你的代码时不会去有意修改你的参数。第二，它告诉了编译器禁止对这个参数做任何修改操作，特别是无意的情况下。如下是来自示例15-3中concat函数的一个不安全的版本：

```
void concatUnsafe(std::string& s1,
                  std::string& s2,
                  std::string& out) {
   out = s1 += s2; // Whoops, wrote to s1
}
```

尽管我有很严谨的编码习惯，但是我还是犯了一个愚蠢的错误，我把+=误用为+了。所以当你调用concatUnsafe函数时，它将修改out和s1两个参数，这可能使得这个函数的用户大吃一惊，因为他/她只希望修改这些源字符串中的一个。

const关键字则可使得这种情况免于发生，重新写一个函数concatSafe，把这些参数声明为const，就像示例15-3中那样，那样编译就不会成功：

```
void concatSafe(const std::string& s1,
                const std::string& s2,
                std::string& out) {
   out = s1 += s2; // Now you will get a compile error
}
```

concatSafe函数保证了s1和s2不会被修改。同时它还传达了一层意思：它只是让用户传递const型的参数。例如，需要连接字符串的代码如下所示：

```
void myFunc(const std::string& s) { // Notice that s is const

   std::string dest;
   std::string tmp = "foo";

   concatUnsafe(s, tmp, dest);  // Error: s is const

   // Do something with dest...
}
```

在这个例子中，myFunc函数不能被编译通过，因为concatUnsafe函数没有维持myFunc函数中参数的常量性。myFunc函数使得它不会修改变量s的内容，这也就意味着任何在myFunc函数中的代码都不能修改s的值。当然，你可以使用const_cast来把这个常量性剔除掉从而使得s可被修改，但是这样做是不好的，你应该避免这样做。concatSafe函数则能编译通过且运行正确。

指针则有点不同。当你把一个指针变量作为参数时，这有两部分的含义：地址本身和地址引用的值。C++能让你使用const来限制这两部分。如下连接函数的另一个使用指针的版本：

www.TopSage.com

```
void concatUnsafePtr(std::string* ps1,
                     std::string* ps2,
                     std::string* pout) {
   *pout = *ps1 + *ps2;
}
```

这和前面的concatUnsafe具有相同的问题。加上const使得目标字符串不会被修改：

```
void concatSaferPtr(const std::string* ps1,
                    const std::string* ps2,
                    std::string* pout) {
   *pout = *ps1 + *ps2;
}
```

很好，你现在不能修改*ps1和*ps2。但是你还是可以修改ps1和ps2，或者使用别的字符串，你可以改变指针的值而不是改变它们指向的字符串，使得它们指向别的字符。这就还是阻止不了你，如下所示：

```
void concatSaferPtr(const std::string* ps1,
                    const std::string* ps2,
                    std::string* pout) {
   ps1 = pout;  // Uh-oh
   *pout = *ps1 + *ps2;
}
```

通过如下这样再次使用 const 则可阻止这类错误：

```
void concatSafestPtr(const std::string* const ps1,
                     const std::string* const ps2,
                     std::string* pout) {
   *pout = *ps1 + *ps2;
}
```

通过在星号的两边都加上const，你就又可使得这个函数变得安全。这就使得你的函数的用户非常清楚地明白你的意图，并且当你犯了一个无意的错误时也能很快就发现。

请参考 15.4 节。

15.4 确保成员函数不会修改其对象

问题

如何调用一个常量型的对象的成员函数，其中你的编译器不允许把这个对象中正在被操作的类型从常量型转换成非常量型类型？

解决方案

在这个类的声明和定义中在这个成员函数的声明处的右边加上 const 关键字。示例 15-4 说明了这种用法。

示例 15-4 声明一个 const 成员函数

```
#include <iostream>
#include <string>
```

www.TopSage.com

大家网 TopSage.com

```
class RecordSet {
public:
   bool getFieldVal(int i, std::string& s) const;
   // ...
};

bool RecordSet::getFieldVal(int i, std::string& s) const {
   // In here, you can't modify any nonmutable data
   // members (see discussion)
}

void displayRecords(const RecordSet& rs) {
   // Here, you can only invoke const member functions
   // on rs
}
```

讨论

在一个成员函数的声明和它的定义的尾部加上const关键字将强制编译器更加仔细地检查这个成员函数体。const常量型成员函数不允许在它的数据成员上面执行一些非常量型的操作。如果这样做了的话，编译器将会报错。例如，如果在RecordSet::getFieldVal成员函数中，我更新了counter成员变量的话，它将不能编译通过（假设getFieldCount_是RecordSet类的一个成员变量）：

```
bool RecordSet::getFieldVal(int i, std::string& s) const {
   ++getFieldCount_;  // Error: const member function can't modify
                      // a member variable
                      // ...
}
```

这将有利于捕获更多的细微错误，因为它的功能跟const关键字修饰一个变量一样（如15.3节那样）。请看如下一个不小心的错误：

```
bool RecordSet::getFieldVal(int i, std::string& s) const {

   fieldArray_[i] = s; // Oops, I meant the other way around
   // ...
}
```

同样的道理，编译器将中断并且抛给你一个错误提示，因为你正企图修改一个成员变量，而这在常量型的成员函数中是不允许的。当然，这也有例外。

在RecordSet类中，就像在示例15-4中那样，你可能想在这个记录集中向前或向后移动，假设这里有一个当前记录的概念。一个简单的办法就是保持一个整型数的成员变量来指示当前记录的下标；你的成员函数可以增加或减小这个下标的值从而使得当前记录向前或向后移动：

```
void RecordSet::gotoNextRecord() const {
   if (curIndex_ >= 0 && curIndex_ < numRecords_-1)
      ++curIndex_;
}

void RecordSet::gotoPrevRecord() const {
   if (curIndex_ > 0)
      --curIndex_;
}
```

www.TopSage.com

很明显，如果这个成员函数是常量型的话，则不能达到这个目的。因为这两个函数都修改了数据成员。但如果不这样做的话，类RecordSet的用户将不能通过RecordSet类的一个const常量型对象来遍历它的记录集。所以在这个const常量型成员函数的规则中也有一个例外，就是C++还支持一个机制：mutable关键字。

如果在类的声明中把curIndex_这个数据成员声明为mutable类型的话，那么在一个const常量型的成员函数中就可以修改这个curIndex_的值：

```
mutable int curIndex_;
```

这就使得你在任何你想修改的地方你都可以传入这个数据成员来进行修改。不过这个规则还是要谨慎地使用，因为它给你的const常量型成员函数开始于非常量型，使得在这一点上与非常量型函数相同了。

就像在示例15-4中那样使用const关键字有效地保证了成员函数不会修改对象的状态。总的来说，它是一个很好的规则，因为它给你的类的用户传达了这个成员函数的行为，并且当你无意中违反了这个规则时，编译器还能强制地发出一个错误信号。

15.5 编写一个不是成员函数的操作符

问题

如何编写一个二进制的操作符，且不让它成为类的成员函数？

解决方案

使用operator关键字、一个临时变量和一个复制构造函数就可以完成它的大部分工作，并返回一个临时对象。示例15-5中说明了给自定义的String实现一个简单字符串的连接操作符。

示例15-5 连接一个非成员操作符

```
#include <iostream>
#include <cstring>

class String {  // Assume the String class declaration
                // has at least everything shown here
public:
   String();
   String(const char* p);
   String(const String& orig);
  ~String() {delete buf_;}

   String& append(const String& s);
   size_t length() const;
   const char* data() const;
   String& operator=(const String& orig);

   // ...
};

String operator+(const String& lhs, const String& rhs) {
```

www.TopSage.com

```
    String tmp(lhs); // Copy construct a temp object
    tmp.append(rhs); // Use a member function to do the real work

    return(tmp);     // Return the temporary
}

int main() {

    String s1("banana ");
    String s2("rancher");
    String s3, s4, s5, s6;

    s3 = s1 + s2;          // Works fine, no surprises
    s4 = s1 + "rama";      // Constructs "rama" automatically using
                           // the constructor String(const char*)
    s5 = "ham " + s2;      // Hey cool, it even does it backward
    s6 = s1 + "rama " + s2;

    std::cout << "s3 = " << s3.data() << '\n';
    std::cout << "s4 = " << s4.data() << '\n';
    std::cout << "s5 = " << s5.data() << '\n';
    std::cout << "s6 = " << s6.data() << '\n';
}
```

讨论

一个独立的操作符的声明和定义基本上和成员函数操作符相同。在示例15-5中，通过如下定义我实现了operator+操作并使它成为一个成员函数：

```
String operator+(const String& rhs);
```

在大部分应用中，无论是你把operator+定义为成员函数还是非成员函数，它们的工作方式基本相同，但还是有很多原因使得你要把它实现为非成员函数。第一个原因是概念上的：让这个操作符返回一个新的相同的对象是否有意义？operator+作为成员函数既不改变对象的状态，同时也不检查对象的状态。它总的来说只是一个操作字符串的函数，所以不应该把它实现为成员函数。

第二个原因就是技术的原因。你不可能用一个成员操作来做如下的工作(来自以上的例子)：

```
s5 = "ham " + s2;
```

它不能工作的原因是一个字符的串没有这个带一个字符串作为参数的operator+操作。另一方面，如果你定义了一个单独的带两个字符串作为参数的operator+操作，你的编译器将把它看作String类的一个带常量型的字符指针参数的（或者任何你正在和String类一起使用的类型）构造函数，并且在运行时构造一个临时对象。因此，以上的代码等同于以下的代码：

```
s5 = String("ham ") + s2;
```

编译器将存储你输入的字符并调用这个操作。

给一个流重载左移和右移操作符（<<和>>）需要你使用非成员操作符。例如，使用左移操作符把一个新对象输出到流中，你就应该如下这样来定义这个<<：

```
ostream& operator<<(ostream& str, const MyClass& obj);
```

www.TopSage.com

当然，你可以继承标准类库中的string类，然后再加上你需要的左移操作符，但这会是一个好的想法吗？你如果这样做的话，只有使用了你的新流的类才能把你自己写的类输出到流中。如果你使用一个单独的操作符，任何在一个命名空间中的代码都可以把你的对象写到输出流ostream中去（或者从输入流istream中读起）。

15.6 初始化由逗号分隔开的数列

问题

如何用一个由逗号分割开的数列来初始化一个序列，就像你能用一个内置的数组来初始化一个序列一样？

解决方案

就像在以下的示例15-6中一样，你可以通过定义一个帮助类并重载逗号操作符，你就可以在这些标准的序列（如矢量和链表）使用逗号初始化语法。

示例15-6 在标准序列中使用逗号初始化语法

```
#include <vector>
#include <iostream>
#include <iterator>
#include <algorithm>

using namespace std;

template<class Seq_T>
struct comma_helper
{
  typedef typename Seq_T::value_type value_type;
  explicit comma_helper(Seq_T& x) : m(x) { }
  comma_helper& operator=(const value_type& x) {
    m.clear();
    return operator+=(x);
  }
  comma_helper& operator+=(const value_type& x) {
    m.push_back(x);
    return *this;
  }
  Seq_T& m;
};

template<typename Seq_T>
comma_helper<Seq_T>
initialize(Seq_T& x) {
  return comma_helper<Seq_T>(x);
}

template<class Seq_T, class Scalar_T>
comma_helper<Seq_T>&
operator,(comma_helper<Seq_T>& h, Scalar_T x) {
  h += x;
  return h;
}

int main() {
  vector v;
  int a = 2;
```

www.TopSage.com

```
  int b = 5;
  initialize(v) = 0, 1, 1, a, 3, b, 8, 13;
  cout << v[3] << endl; // outputs 2
  system("pause");
  return EXIT_SUCCESS;
}
```

讨论

通常一个标准序列的初始化是通过多次调用push_back成员函数来进行的。由于这样的工作有点重复，所以我写了个函数：initialize函数，它是实现了用逗号初始化从而减少了这种重复性的工作。

你可能没有意识到逗号其实是一个操作符，是一个可以被重载的操作符。有很多人也跟你一样；因为逗号作为操作符这样使用并不是很常见。但作为这样的初始化目的，逗号还是可以被重载的。

这个方案使用了一个helper类的函数initialize，它返回一个helper类的模板:comm._helper。这个helper类模板保持一个指向这个序列的索引，并且重载了逗号(,)操作符、赋值(=)操作符和+=操作符。

因为编译器读赋值语句“v = 1, 1, 2, …”的方式，所以在这个方案中定义了一个独立的helper类函数；由于标准序列不支持从一个简单的值上赋值，所以编译器将会把“v = 1”认为不合法。Initialize函数所做的就是构造一个合适的comma_helper类的对象，这个对象持有一个序列并且重载赋值和逗号操作符。

逗号操作符也可称为构造序列操作符，它的默认功能就是从左到右地构造表达式，并且与最右边的值具有相同的类型和值。当重载以后，这个操作符就具有新的含义，而原来默认的语义就不存在了，并且不能再保证从左到右地读取参数，示例15-7中的代码也许与你期望的可能不同。

示例15-7 重载逗号操作符参数的求值顺序是没有定义的

```
int prompt_user() {
   cout << "give me an integer ... ";
   cin >> n;
   return n;
}

void f() {
   vector<int> v;
   // The following could result in v being initialized out of
   // sequence
   intialize(v) = prompt_user(), prompt_user();
}
```

正确地写函数f的办法应该是把每一个prompt_user函数都放在一条独立的语句中。

注意：Thorsten Ottosen的Boost赋值类库支持在标准集合上更为完全的逗号序列初始化，也支持一些其他形式的初始化。在*http://www.boost.org*网站上可找到这个库。

www.TopSage.com

作者简介

大家网
TopSage.com

D. Ryan Stephens 软件工程师和作者，居住在Arizona的Tempe。他乐于进行各种语言，尤其是C++语言的程序设计。他的研究兴趣包括信息检索与数据挖掘，以及有关算法和大型数据集的研究。当他不工作、不写作或编程时，喜欢和他的儿女们戏玩，在他的房子周围劳作，或骑自行车出去游玩。

Christopher Diggins自由的软件开发人员和作者，当他还是孩子的时候，就开始进行计算机程序设计。Christopher定期为C++ Users Journal写作，是Heron程序设计语言的设计者。

Jonathan Turkanis是Boost Iostreams库和其他几个开源C++库的作者，研究领域包括智能指针、运行时映像、组件体系结构和面向方面程序设计。他是美国加利福尼亚伯克利大学的博士生。

Jeff Cogswell软件工程师，居住在Ohio的Cincinnati。他从C++语言发明后就开始使用它进行程序设计，已编写了两部C++的图书。他也喜欢用其他语言编程，尤其是Python。不工作时，喜欢阅读小说或弹奏吉他。

封面介绍

封面上的动物是牧羊犬。它的名字表明，这是一种帮助牧羊人放牧的狗，起源于17世纪的苏格兰和英格兰的高原地带。苏格兰高地的绵羊的腿上和脸上有黑色的毛，称为“Colley”绵羊，这个名字从老苏格兰语的“coal”演变而来。现代的牧羊犬比其苏格兰祖先颜色更浅、骨骼更粗壮，于19世纪后期开始在英格兰繁殖。现在，牧羊犬主要成了家庭宠物，但在美国仍用作农场犬。

有两种不同的牧羊犬：毛皮粗糙的用于看护绵羊；毛皮光滑的用于驱赶牲畜去市场。这两种犬都是灵活的、有着流行型体形的狗，有着锐利的嘴和敏锐的眼睛。它们一般为22~26英寸高，50~75磅重。它们的毛通常为白色，另一种颜色是黄白色、褐白色，甚至是炭黑色。

www.TopSage.com